GENERATION AND USE OF A DISCRETE ROBOTIC CONTROLS ALPHABET FOR HIGH-LEVEL TASKS

A Thesis
Presented to
The Academic Faculty

by

Eugene F. Gargas III

In Partial Fulfillment
of the Requirements for the Degree
Masters of Science in the
School of Electrical Engineering

Georgia Institute of Technology
May 2012

GENERATION AND USE OF A DISCRETE ROBOTIC CONTROLS ALPHABET FOR HIGH-LEVEL TASKS

Approved by:

Professor Magnus Egerstedt, Advisor
School of Electrical Engineering
Georgia Institute of Technology

Professor Ayanna Howard
School of Electrical Engineering
Georgia Institute of Technology

Professor Yorai Wardi
School of Electrical Engineering
Georgia Institute of Technology

Date Approved: 27 March 2012

To my family,

for supporting me in

all my endeavors

ACKNOWLEDGEMENTS

I would like to thank my adviser Dr. Magnus Egerstedt for guiding me through the exact research experience I was looking for when I came to Georgia Tech. I'd also like to thank the members of the GRITS lab for being there to bounce ideas off of, and participating in my experiment. Finally, I'd like to thank my family and friends for their support while I've been completing my Master's degree.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The objective of this thesis is to generate a discrete alphabet of low-level robotic controllers rich enough to mimic the actions of high-level users using the robot for a specific task. This alphabet will be built through the analysis of various user data sets in a modified version of the motion description language, MDLe. It can then be used to mimic the actions of a future user attempting to perform the task by calling scaled versions of the controls in the alphabet, potentially reducing the amount of data required to be transmitted to the robot, with minimal error.

In this thesis, theory is developed that will allow the construction of such an alphabet, as well as its use to mimic new actions. A MATLAB algorithm is then built to implement the theory. This is followed by an experiment in which various users drive a Khepera robot through different courses with a joystick. The thesis concludes by presenting results which suggest that a relatively small group of users can generate an alphabet capable of mimicking the actions of other users, while drastically reducing bandwidth.

CHAPTER I

INTRODUCTION

There are many forms of robotic implementations, ranging from autonomous rovers and swarm networks to haptic interfaces and much more. These implementations require a control structure which falls into one of the three general forms of robotic architectures: the deliberative approach, the reactive approach, or the hybrid approach.[2] And in each of these architectures there exists a layer that calculates the low-level control signals that exact the desired physical change on the robotic system.

In a reactive approach, these control signals are based purely on the state of the robot within its environment. But in the hybrid and deliberative approaches, these control signals are based on the wishes of some high-level planner attempting to exact a coordinated movement in the world.

Unfortunately, the calculation of these control signals are highly dependent on the exact robot we are implementing the system on. But a high-level planner should not have to be fine tuned for the low-level control signals. For instance, if a path planner wants to go forward, it should only have to send the command 'go forward', regardless of whether it has wheels or legs.

So it is desirable to create a degree of modularity between the high-level algorithms/users which are concerned with tasks like navigation, and the low-level controllers concerned with turning such commands as 'go forward' into the exact DC motor reference values that get the wheels on the robot to go forward. This idea is shown in figure 1.

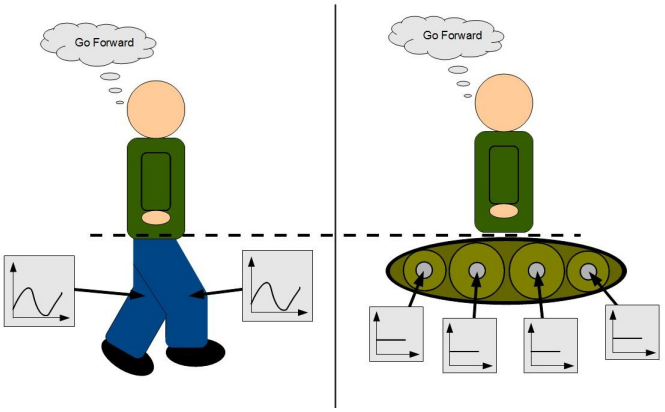


Figure 1: Basic Concept of a Motion Description Language

1.1 Motion Description Languages

In the late 1980's Roger Brockett developed a 'Motion Description Language' (MDL) that allowed for this form of modularity. His idea was to store in memory the low-level controls required to perform high-level tasks on a particular robotic apparatus, and then create a mapping function between the high-level commands and the stored controls that allow the robotic apparatus to perform the command. Now, if the user wishes to use the high-level planner on a different apparatus, all the designer has to do is figure out the new control required to perform the task, and re-map the high-level command to this new control.

So MDL is a language for compartmentalizing the low-level controls of a specific apparatus for use with various high-level planners. But it relies on the fact that for each robotic apparatus you know exactly which controls to store that will allow the high-level user to accomplish their tasks, which implies that you know every task the apparatus will be used for. This seems unlikely to me, and so we ask, how can we build a set of controls that will allow the high-level systems to accomplish all of their tasks?

The answer I believe, is through bottom-up mimicry. Imagine that you, and everyone else is a robot. Now suppose we have a pair of robotic legs that can be

interchanged between members of the community, and can perform whatever action the user wants. Some users may run marathons, some may swim, some will do flips and cartwheels, while some will just sit. There are too many possibilities for an MDLe designer to foresee when attempting to design an all-encompassing set of controls to store on our robotic legs. So instead, lets just record all of the control signals that allowed the performance of all the actions the users perform. And when we reach a point after which we feel that most actions the legs may ever be asked to perform, have already been performed, we can analyze the set of controls we recorded and develop a base set of controls which, through some form of combination, can mimic every single control that was recorded. This set of controls would truly be a proper language that wouldn't inhibit any of the higher-level users in their tasks.

1.2 Goals of the Thesis

In this thesis I will develop a theory and algorithm to build this base set of MDL controls using the method of bottom-up mimicry discussed at the end of the previous section. Once this is accomplished, I will make an attempt at actually building a set of controls through an experiment in which I get a large set of different users to use a robot to accomplish a few tasks in any manner they wish. We will then see if the developed set of controls is rich enough to allow a new set of users to use the robot in any manner they wish. If successful, it would mean that we have in fact developed a method of building an MDL controls set from the ground up that is broad enough to accommodate any user's task.

As a secondary goal, I would like to determine if this method of using MDL has any other benefits than modularity. I suspect that having the control signals stored on board and only calling them by name would greatly diminish the bandwidth on the bus connecting the high and low level systems. This could be very useful for haptic interfaces, or extra-planetary robots in which the communications link between the

high-level controller and low-level system has limited bandwidth.

CHAPTER II

BACKGROUND AND PREVIOUS WORK

In this chapter I will introduce the various forms of motion description languages that have been developed over the years. We will begin with the original MDL specification mentioned in the introduction, and then discuss the successor, MDLe. MDLe will be the basis for a custom MDL I develop in chapter 3 that will allow us to build a generically rich set of controls. We will conclude with some examples of MDLe in action, as well as a coverage of the relatively minute amount of previous work on generic alphabet construction.

2.1 MDL

In the late 1980's, R. Brockett formalized a language he called MDL, or 'Motion Description Language'. The overall purpose of this framework was to provide a clear separation between the low-level controller and the high-level user [3]. Its implementation relied on storing a set of control signals needed to perform very specific segments of a high-level motion, and then creating a map between the high-level commands and the stored signals. Then by calling these signals sequentially, the user could enact the high-level motions. The reasoning for MDL's implementation came from an intuitive understanding of how we as humans operate.

To begin, let us define a model for the lowest-level description of hardware for motion control.

$$\dot{x}(t) = f(x(t)) + G(x(t))v(t)\ ;\ y(t) = h(x(t))$$

with x being an n-dimensional state vector, v being a m-dimensional control vector, and y being a p-dimensional vector consisting of the sensor outputs.

If we let our control $v(t) = u(t) + k(y(t))$, with $u(t)$ being an open-loop control, and $k(y(t))$ a feedback law, then our hardware description becomes

$$\dot{x}(t) = f(x(t)) + G(x(t))(u(t) + k(y(t))) \; ; \; y(t) = h(x(t)) \tag{1}$$

This statement is a useful description of how the hardware will evolve over time given an original state x_0, and open-loop control $u(t)$. Unfortunately, its usage does not correspond to our everyday experience.

"One can easily appreciate that many human motions appear to be a combination of a pre-positioning phase achieved with little or no feedback, followed by an environmental adjustment phase characterized by a significant use of feedback. Walking, grasping and shaking hands are examples of what we mean. Thus it seems that in dealing with complex systems it is more insightful to think of the mode of control as alternating between these two possibilities." [3]

In terms of (1), we can model this as a sequential concatenation of state evolutions corresponding to different combinations of open-loop controls $u(t)$ and feedback laws $k(y(t))$. This leads to a formal definition of MDL.

Let us define a triple (u, k, T) in which u is an open loop control, k is a feedback law, and T is an epoch of time. We refer to these triples as *modal segments* because they specify a mode of control over a segment of time. If we pass the string of modal segments $(u_1, k_1, T_1)(u_2, k_2, T_2)...(u_r, k_r, T_r)$, then our MDL device will execute a motion defined by

$$\begin{aligned}
\dot{x}(t) &= f(x(t)) + G(x)(u_1(t) + k_1(y(t))) \quad 0 \le t < T_1 \\
\dot{x}(t) &= f(x(t)) + G(x)(u_2(t) + k_2(y(t))) \quad T_1 \le t < T_1 + T_2 \\
&\vdots \\
\dot{x}(t) &= f(x(t)) + G(x)(u_r(t) + k_r(y(t))) \quad \sum_{i=1}^{r-1} T_i \le t < \sum_{i=1}^{r} T_i
\end{aligned}$$

This definition of MDL was the beginning of a universal robotics language capable of separating the higher-level tasks from the lowest levels of control.

2.2 MDLe

In the mid-90's, a few improvements to Brockett's MDL were introduced. The first aided researchers working with reactive planners wishing to implement MDL on autonomous robots. These robots operate in environments in which they have little to no *a priori* information, which can cause the need for a sudden switching of controls so as to avoid catastrophic failures, like when a rover moving forward encounters a cliff. Unfortunately, MDL provided no means of ending a modal segment other than the expiration of the time period T. So an interrupt based on sensory information was incorporated that would allow the sudden interruption of an executing call.

A second improvement is of direct interest to us. Imagine we have a pair of bionic legs that we want to store a single 'move forward' control signal on. MDL would not allow this because a single step forward for a human is actually a sequence of n even more fundamental motions, for which the open-loop controls and feedback laws differ. So to get the robot to move forward indefinitely, we would need to keep calling these n modal segments for all time. It would be much easier for a high-level system to group these n modal segments into one behavior, and then call that single behavior instead. And so this grouping of controls into a single 'behavior' was added into the specification of MDLe. [10][11][12]

2.2.1 Formal MDLe Definition

I will now proceed to offer the formal definition of MDLe as specified in [12], as it is a strong basis for the work of my thesis.

Let us redefine the general hardware description (1) proscribed in MDL as

$$\dot{x}(t) = f(x(t)) + \sum_{i=1}^{m} g_i(x)u_i(t) \; ; \; y = h(x) \in \mathbb{R}^p$$

where

$$\begin{aligned} x(\cdot) &: \mathbb{R}^+ = [0,\infty) \to \mathbb{R}^n \\ u_i &: \mathbb{R}^+ \times \mathbb{R}^p \to \mathbb{R} \\ & \quad (t, y(t)) \mapsto u_i(t, y(t)) \end{aligned}$$

Further, each g_i is a vector field in $\mathbb{R}^n$

We define the **atom** of MDLe in a manner analogous to the modal segment (u, k, T) defined in MDL. Let each atom, denoted by σ, be a triple of the form $\sigma_i = (U_i, \xi_i^a, T_i^a)$ where

$$U_i = (u_1, \ldots, u_m)$$

Here, each u_j is a control, with the boolean function ξ_i,

$$\begin{aligned} \xi_i^a &: \mathbb{R}^k \to \{0,1\} \\ & \quad s(t) \mapsto \xi^a(s(t)) \end{aligned}$$

the time period $T_i^a \in \mathbb{R}^+$, and a k-dimensional signal representing the output of k signals, $s(\cdot) : [0,T] \to \mathbb{R}^k$.

The value ξ_i^a can be interpreted as an interrupt to the system, introduced to accommodate the need of reactive planners wishing to prevent a catastrophic failure, or adapt to a changing environment.

Let us denote $\widehat{T}_a$ (measured with respect to the initiation of the atom) the time at which an interrupt was received, i.e. ξ^a changes from 1 to 0.

If we call the string of atoms

$$\sigma_1 \cdots \sigma_n = (U_1, \xi_1^a, T_1^a) \cdots (U_n, \xi_n^a, T_n^a)$$

then our state will evolve according to

$$\begin{aligned} \dot{x}(t) &= f(x(t)) + G(x(t))U_1 \quad 0 \le t < min[\widehat{T}_1^a, T_1^a] \\ &\vdots \\ \dot{x}(t) &= f(x(t)) + G(x(t))U_n \quad \sum_{i=1}^{n-1} min[\widehat{T}_i^a, T_i^a] \le t < \sum_{i=1}^{n} min[\widehat{T}_i^a, T_i^a] \end{aligned}$$

where $G(x) = (g_1(x) \cdots g_m(x))$

Hence, each atom in the input string is executed in sequential order, with the execution of each atom being inhibited via interrupts or a "time-out" via the timer T_i^a.

These low-level definitions resolve the problem faced by reactive planners, but do nothing to address the desire to group atoms together. For this we must define the higher-level structure of MDLe.

Let us first define a scaled atom.

Definition 2.2.1. Given an atom (U, ξ^a, T^a), define

$$(\alpha U, \xi^a, \beta T^a), \quad \alpha = (\alpha_1, \ldots, \alpha_m) \in \mathbb{R}^m, \quad \beta \in \mathbb{R}^+$$

as the **scaled atom** denoted by $(\alpha, \beta)(U, \xi^a, T^a) \to (\alpha, \beta)\sigma$

We see that α is used to spatially scale the control in an atom, while β scales the length of time for which each atom is executed.

Let us now define an alphabet to contain our atoms.

Definition 2.2.2. An **alphabet** Σ is a finite set of independent atoms. Thus $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ for some finite n where σ_i denotes the triple (U_i, ξ_i^a, T_i^a), such that $\sigma_i \neq (\alpha, \beta)\sigma_j$ for some $\alpha \in \mathbb{R}^m, \beta \in \mathbb{R}^+$ and $i = 1, \ldots, n, j = 1 \ldots n, i \neq j$. Hence, an alphabet is a set of atoms for which none can be derived from other atoms.

Definition 2.2.3. An **extended alphabet** Σ_e is the infinite set of scaled atoms derived from Σ.

Definition 2.2.4. A **language** Σ^* (or respectively Σ_e^*) is defined as the set of all strings over the fixed alphabet Σ (or Σ_e).

With this high-level structure now defined, we conclude with the extension allowing for the specification of a behavior, or compilation of atoms in Σ_e^*. To simplify notation, we denote the scaled atom $(1, 1)\sigma_i$ by σ_i.

Definition 2.2.5. A **behavior**, denoted by π, is an element(or word) of the extended language Σ_e^*, with an associated timer T^b and interrupt ξ^b. For example, given an alphabet $\Sigma = \{\sigma_1, \sigma_2\}$, a behavior π_i could be

$$\pi_i = ((\alpha_{i1}, \beta_{i1})\sigma_{i1}, (\alpha_{i2}, \beta_{i2})\sigma_{i2}, (\alpha_{i3}, \beta_{i3})\sigma_{i1}, \xi^b, T^b)$$

with σ_{ij} denoting the j^{th} atom in the i^{th} behavior, and α_{ij}, β_{ij} corresponding to the scaling factors of the j^{th} atom in the i^{th} behavior.

Finally, interrupts associated with atoms (ξ^a) are level-0 interrupts, while interrupts associated with behaviours (ξ^b) are called level-1 interrupts. If a level-0 interrupt is received, the next atom in the executing behaviour will be run. If a level-1 interrupt is received, the next behaviour will be run. Likewise, the T^a specifies the time-out of each atom, while T^b specifies the time-out of a behaviour.

2.2.2 MDLe applications

Since its formalization, MDLe has been applied in a variety of applications. One common use is by path planners for wheeled robots exhibiting unicycle dynamics, such as in [12] and [7]. This is common because the full spectrum of motions available to these robots is relatively small, and can be intuitively defined in MDLe.

Yet there are other non-standard applications for which MDLe has been shown to be effective. One instance is for use by robotic marionettes to put on plays. As can be seen in [17][14][16], the manner in which the manipulator of a marionette produces a play is in a sequential concatenation of various moves, which lends itself nicely to be implemented as an MDLe plan. An example of a typical play is shown in figure 2.

In fact, MDLe has even been applied towards modeling biological systems. In [12] there is an interesting example of modeling a frog's predator/prey response. Typically, a frog will switch back and forth between predator mode, in which it approaches small animals based on a detection signal issued by the tectum region of the brain, and a

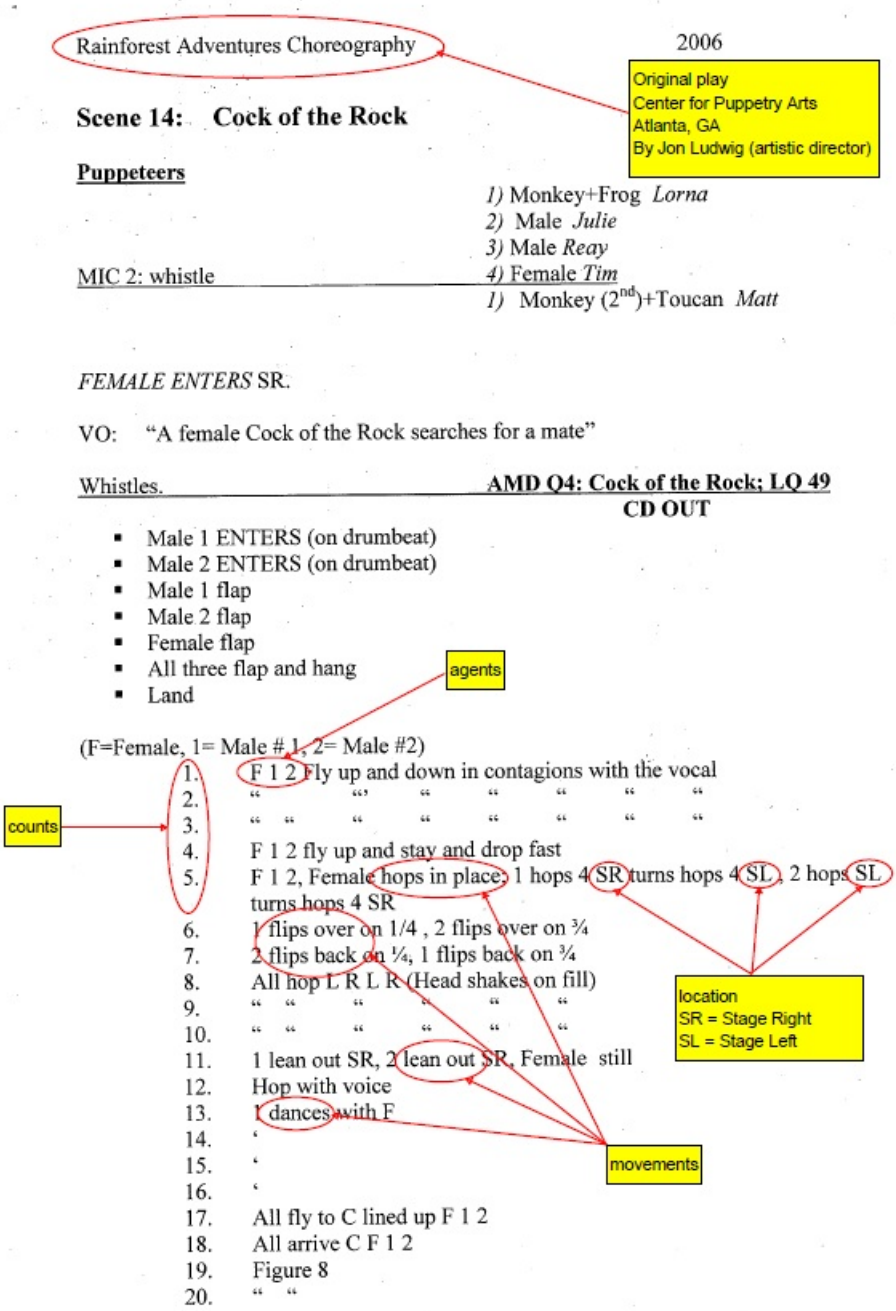


Figure 2: Typical Marionette Play

prey response, in which it retreats from large animals based on a detection signal from the pretectum region of the brain. These behaviors can be modeled as two MDLe atoms: approach and retreat, with the brain signals acting as interrupts. Based on observations that frogs with a damaged pretectum will approach both large and small animals, with no retreat response, it was shown that a frog operates on the infinite MDLe string

$$\sigma_{approach}\sigma_{retreat}\sigma_{approach}\cdots$$

I direct the interested reader to page 206 in [12] for a proof of this.

2.2.3 MDLe Alphabet Construction

So as we see, MDLe has reached a level of formalism that is adequate for many researchers, and gives us the tools we need to store the controls capable of enacting high-level motions, as was shown in the examples of the previous section. But again the underlying assumption made in all of these examples was that we knew what these high-level motions were to begin with. So how do we develop a generic MDLe alphabet rich enough to produce all motions desired by the high-level system?

There has been substantially less research into this question than on how to actually apply MDLe. But there have been a few notable pushes.

One formulation sets the alphabet as a set of constant controls. They then employ lattice theory to determine when and how to switch controls. [6] This solution involves a lot of overhead though, and it would be preferable to store more complex controls.

Another push was made in [19][18] in which the authors identified commonly used sequences of MDLe atoms, and combined them into a single new atom using optimal control techniques. Thus, in essence, they were creating the fundamental controls most commonly required by the system. The one underlying assumption though was that the alphabet was initially populated with a few controls tailored to the system.

And so we arrive at the topic of my thesis. How do we produce an alphabet

capable of producing all of the motions it may be asked to perform?

CHAPTER III

THEORY

In this chapter I will present a mathematical foundation for building an all-encompassing controls alphabet within a new MDLe framework using the bottom-up mimicry method discussed in chapter 1. This alphabet will be comprised of both MDLe atoms and behaviors. We will also develop a method to decompose any given control signal into an MDLe string based on these alphabets.

We will begin with a discussion on how to approach the task of building a descriptive MDLe alphabet from a given control signal, resulting in a re-specification of MDLe that will allow us to scale and combine atoms in a new manner. Within this re-specification, we will then develop a theory on how to build and use a low-level atom alphabet, followed by a theory on how to build and use a behavioral alphabet based on the atom alphabet.

3.1 Framing an Approach

To begin, we must first investigate the question of how we can build an alphabet from scratch given a control signal used by a robotic apparatus to perform some high-level task. In order to properly approach this question, we must re-familiarize ourselves with how we are trying to build this alphabet.

As defined earlier in section 2.2.1, MDLe allows us to sequentially call stored control atoms. A control signal generated by these MDLe calls may look something like that shown in figure 3.

Now our goal is to build an MDLe alphabet using the method of bottom-up mimicry. So our task is to take the signal of figure 3, and work backwards to define a set of atoms that can produce it.

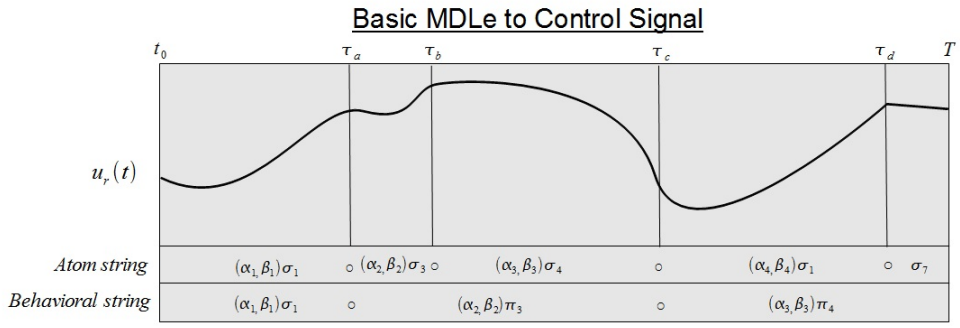


Figure 3: Possible Control Signal from MDLe Strings

So the first thing we need to think about is how we can transform a given control signal into the language of MDLe, so that we can then identify atoms that need to be stored in our controls alphabet.

3.1.1 Temporal Segmentation

The first step we can take is to put a given control signal into the most basic structure of MDLe: a string of sequential atoms.

To demonstrate a naive method of doing this, lets define a time period $\triangle t = 1s$ and the points $\tau = [\triangle t, 2 \triangle t, \ldots, T]$, which can be used to cut the signal into n distinct segments ordered sequentially. By defining each signal segment as an atom $\sigma_i,\ i \in [1, n]$ and calling the MDLe string

$$\sigma_1 \circ \sigma_2 \circ \ldots \circ \sigma_n$$

we will get a control signal that is an exact replica of the original signal.

Unfortunately, this approach would likely lead to a huge alphabet of atoms whose only relation to the high-level task depends on where in the signal they occurred with respect to the original start time.

For a more elegant solution, we remember that these signals exist to enact motion in the real world. So lets return to the notion that Brockett initially used when

developing MDL, in that motion is more like a concatenation of switching control modes.

Take a moment and think about how you perform a handshake, purely in terms of the forward motion of your arm. The first step is to accelerate your arm towards the person. This is followed by a gliding period in which your arm moves towards the person with a constant velocity. Then to finish, you decelerate your arm to the actual grasp of the others hand. Mathematically this looks like the signals in figure 4.

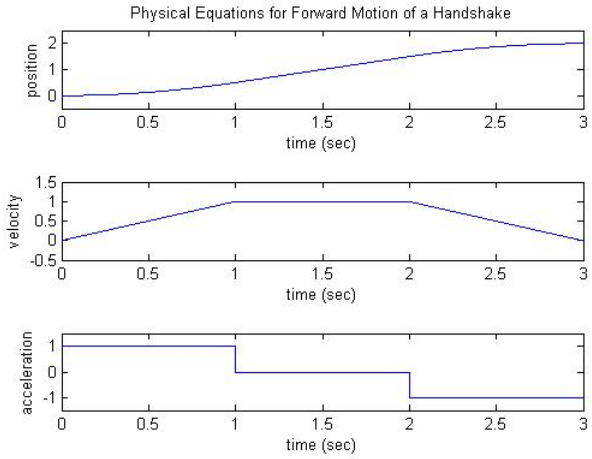


Figure 4: Forward Motion for a Handshake

If we look in the sub-graph at the bottom of figure 4, which represents the acceleration of your arm to perform the handshake, we see that the boundaries between the 'fundamental' tasks I defined are accompanied by major changes in acceleration. So lets use this observation as a basis for how to segment our signal.

Given a control signal $u_r(t)$, we can find the first and second derivatives of the

signal. We then define a discontinuous function $d(t)$ as follows.

$$d(t) = \begin{cases} 1 \quad if \quad \begin{cases} \ddot{u}_r(t) = 0 \text{ and } \ddot{u}_r(t-\epsilon) \neq 0 \\ \ddot{u}_r(t) \neq 0 \text{ and } \ddot{u}_r(t-\epsilon) = 0 \end{cases} \\ \\ 0 \quad if \quad \begin{cases} \ddot{u}_r(t) \neq 0 \text{ and } \ddot{u}_r(t-\epsilon) \neq 0 \\ \ddot{u}_r(t) = 0 \text{ and } \ddot{u}_r(t-\epsilon) = 0 \end{cases} \end{cases}$$

where $\epsilon \to 0$. Intuitively, the points in time for which $d(t) = 1$ correspond to pure inflection points of the signal, or the boundaries between when a signal enters or leaves a period of zero acceleration. If we define a vector τ filled with all of the times for which $d(t) = 1$.

$$\tau = [\tau_1, \ldots, \tau_n]$$

we can then proceed to segment the signal into n sequential signals with their temporal boundaries on the times declared in τ.

And if we define each of these n segments as an atom of the form

$$\sigma_i = (u_r(t), \xi_i, T_i) \ , \ t \in [\tau_{i-1}, \tau_i]$$
$$st \ \ T_i = \tau_i - \tau_{i-1}$$

then an MDLe call of the form

$$\sigma_1 \circ \cdots \circ \sigma_n$$

will proceed according to

$$\begin{aligned} u_{MDLe}(t) &= u_1(t) & 0 \leq t < \tau_1 \\ u_{MDLe}(t) &= u_2(t - \tau_1) & \tau_1 \leq t < \tau_2 \\ &\vdots & \\ u_{MDLe}(t) &= u_n(t - \tau_{n-1}) & \tau_{n-1} \leq t < \tau_n \end{aligned}$$

and will be an exact replica of the original reference control.

This formulation gives us a good method on how we can define MDLe atoms based on a given control signal.

3.1.2 Framework for an Alphabet

With the segmentation method defined in 3.1.1, we now have a basic tool allowing us to transform any given control signal into a sequence of signals that we can generally think of as MDLe atom calls. So our next task should be to use those signal segments to physically construct an atom alphabet, denoted as Σ_a.

Your first thought might be to just use each of these signals as an atom. But by the definition of an MDLe alphabet in section 2.2.1, there can be no set of scalars $\{\alpha, \beta\}$ for which any atom in Σ_a can be constructed from another.

So in building our alphabet, we must only define atoms that are not scaled versions of other atoms. Further, we would like to build the alphabet in a manner that will minimize the number of atoms defined, yet still have the breadth to recreate the maximum number of signal forms. These stipulations sound like the characteristics of a vector space.

Suppose we were to define a space in which each basis vector was an atom in our alphabet. By the very definition of a basis vector, it would guarantee that none of the other atoms were scaled copies of each other. Further, if we were to incorporate some form of parallel addition between the atoms, by definition of a basis again, our set of atoms could be combined to create any signal segment that lied in the *span* of our alphabet.

But how can we create such a vector space from our atoms? Well, by the segmentation method defined in 3.1.1, we know that our atoms will all be signal segments of the form

$$u(t)\ ,\ t \in [0, T]$$

Lucky for us, there exists such a vector space that allows each vector to be defined as a signal on the time frame $t \in [0, T]$. This is the $L_2[0, T]$ Hilbert space.

3.1.2.1 Alphabet as a Hilbert Space

The $L_2[0,T]$ Hilbert space is a function space in which a vector is defined as a measurable signal on the time scale of $t \in [0,T]$. The space is also equipped with an inner product of the form:

$$\langle u, v \rangle = \int_0^T u(t)v(t)dt$$

Now, the beauty of the Hilbert space is that it allows us to use all of the fundamental concepts of $\mathbb{R}^n$ vector spaces, but with signals scaled to the same time frame. But how can we use this as a framework for our alphabet Σ_a?

Assume we have segmented a given control signal into n distinct signals $u_i(t),\ t \in T_i$, that cannot be made from scaled versions of each other. We can now define a time period T, and find the scalars $[\beta_1, \ldots, \beta_n]$ such that $\beta_i = \dfrac{T_i}{T}$ $, \forall i \in [1,n]$. Now, the signal segments can be temporally compressed or dilated so that all $u_i(\beta_i t)$ exist on the same time frame $t \in [0,T]$. This implies that they also exist in the same Hilbert space H.

Let us now create a subspace $U \subseteq H$ which contains all of the scaled signals $u_i(\beta_i t)$. By assuming that none of these signals were scaled versions of each other, we know $\nexists \alpha_i$ st $\alpha_i u_i(\beta_i t) = u_j(\beta_j t)\ \forall i,j \in [1,n]$ $, i \neq j$. This means that in U, the signals are linearly independent, and thus form a basis for U. So if we proceed to only define atoms in Σ_a that are linearly independent signals $u_i(\beta_i t)$, we can guarantee that the alphabet is valid.

Further, with the alphabet defined as the basis vectors of a space, we are given a powerful new tool in the form of vector addition. By employing vector addition as a means of combining atoms in parallel, we can create infinitely many new signals that are not defined as atoms of Σ_a, but lie in the span of the atoms in our alphabet.

An even greater benefit is that if we can now add our atoms together, we can interact with the alphabet through the projection theorem. This will allow us to quickly identify whether a new signal segment $u_r(\beta_r t)$ can be made using an existing

Σ_a, and if so, which parallel combination of existing atoms to use. Further, if it doesn't exist in the space defined by Σ_b, we can quickly augment the space with a new signal, allowing us to span an even greater space of possible signals.

3.1.3 Re-specification of MDLe

So we see that structuring our atom alphabet as an $L_2[0,T]$ space provides many benefits. By only adding atoms to the alphabet that act as basis vectors of this space, we guarantee that the alphabet is valid. Further, using vector addition with these atoms allows us to generate signals not defined in the alphabet, and affords us the use of the projection theorem to interact with the alphabet.

Unfortunately, to generate an $L_2[0,T]$ space, we need a method to temporally compress or dilate our signal segments to the time frame $t \in [0,T]$, which MDLe does not afford. And even if it did, MDLe offers no operator that can emulate the vector addition between atoms, which means no projection theorem.

Well, there is no law which says we have to conform to the current specification of MDLe. So lets alter the specification in a way that allows us to temporally scale complete signals to a new time frame, and gives us the addition operator we need to use the projection theorem.

3.1.3.1 The Scaled Atom

The first thing we need to redefine is what it means to spatially and temporally scale an atom.

In the current MDLe specification, we can scale an atom according to the following rule

$$(\alpha,\beta)\sigma_i = (\alpha u_i(t), \xi, \beta T_i) \rightarrow \ \alpha u_i(t)\ , t \in [0, \beta T_i]$$

As you can see, the temporal scalar β does not have the desired effect of compressing or dilating the full signal. Rather, it just cuts the signal off at a different point in time, which may lie outside of the temporal range σ_i is defined on.

To fix this, and accomplish our desired version of spatial and temporal scaling, we will define a new form of scaled atom.

Definition 3.1.1. A **scaled atom** scales a signal spatially by scalar multiplication, and temporally by compressing or dilating the full signal to a new time period $[0, T]$.

$$(\alpha, \beta)\sigma_i = (\alpha u_i(\beta t), \xi, \frac{T_i}{\beta}) \rightarrow \ \alpha u_i(\beta t)\ , t \in [0, T]$$
$$\text{st } \beta = \frac{T_i}{T}$$

We see that this definition accomplishes our goal of producing a temporally scaled version of the signal on the time scale $t \in [0, T]$. This definition of a scaled atom will be used for the rest of the thesis.

3.1.3.2 The Merge Operator and Merged Atom

With the new definition of the scaled atom 3.1.1, we have granted ourselves the ability to scale a signal so that it can exist in an $L_2[0, T]$ Hilbert space. Now we need the ability to combine these scaled signals through vector addition, allowing us to create the maximum number of possible signals from our alphabet Σ_a.

Therefore, I would like to introduce the MDLe merge operator, which emulates vector addition between our atoms.

Definition 3.1.2. The **Merge Operator** combines atoms that exist in a common $L_2[0, T]$ Hilbert space using vector addition.

$$\begin{aligned}(\alpha_1, \beta_1)\sigma_1 || (\alpha_2, \beta_2)\sigma_2 &= (\alpha_1 u_1(\beta_1 t) + \alpha_2 u_2(\beta_2 t), min\{\xi_1, \xi_2\}, T) \\ \text{st } T &= \frac{T_1}{\beta_1} = \frac{T_2}{\beta_2}\end{aligned}$$

For simplicity of notation in later sections, we will now go one step further and define a merged atom, which is a signal in the $L_2[0, T]$ space existing as the addition of the atoms in Σ_a, but is not a member of Σ_a. This form of atom will <u>not</u> exist as a part of the alphabet Σ_a, but is defined only for ease.

Definition 3.1.3. A **Merged Atom** is a merged set of scaled atoms that exist in a common $L_2[0,T]$ Hilbert space, but is not a member of Σ_a.

$$\tilde{\sigma}_i = (\alpha_{i1}, \beta_{i1})\sigma_1 || (\alpha_{i2}, \beta_{i2})\sigma_2$$

3.1.3.3 The Scaled Behavior

In later sections, we will want to find behaviors. But as we have altered the definitions MDLe uses to define a behavior, we will need to redefine it here. The concept will be similar to our atoms though, in that we will just be working with scaled signal segments.

The first step we will need is the ability to scale these merged atoms so that they exist on a new time frame.

Definition 3.1.4. A **Scaled Merged Atom** is a merged atom that is spatially scaled by a scalar α_b, and temporally scaled to a new time frame by β_b.

$$\begin{cases} (\alpha_b, \beta_b)\tilde{\sigma}_i &= (\alpha_b\alpha_{i1}, \beta_b\beta_{i1})\sigma_1 || (\alpha_b\alpha_{i2}, \beta_b\beta_{i2})\sigma_2 \\ &= (\alpha_b\alpha_1 u_1(\beta_b\beta_1 t) + \alpha_b\alpha_2 u_2(\beta_b\beta_2), min\{\xi_1, \xi_2\}, T) \\ \text{st } T &= \dfrac{T_1}{\beta_b\beta_1} = \dfrac{T_2}{\beta_b\beta_2} \end{cases}$$

With this definition, we can now make the re-definitions of a behavior. In the original MDLe specification, a behavior is defined as follows

$$\pi_i = ((\alpha_1, \beta_1)\sigma_1 \circ \cdots \circ (\alpha_n, \beta_n)\sigma_n)$$

And can be thought of as a signal, comprised of sequentially concatenated signals related to atom calls.

But we will redefine it to be a sequential combination of the merged atoms defined in 3.1.3. This can be rewritten as follows.

Definition 3.1.5. A **behavior** is a sequential concatenation of merged atoms.

$$\pi_i = (\tilde{\sigma}_1 \circ \cdots \circ \tilde{\sigma}_n)$$

We can think of this behavior as a signal that exists on the time frame $t \in [0, T_i]$. So if we think of it like that, there is no reason we can't scale it again both spatially and temporally to a new time period T. Doing this gives us a definition for the scaled behavior.

Definition 3.1.6. A **scaled behavior** is a behavior spatially scaled by the scalar α_b, and temporally scaled to a new time frame $[0, T]$ by the scalar β_b.

$$\begin{cases} (\alpha_b, \beta_b)\pi_i &= (\alpha_b, \beta_b)(\tilde{\sigma}_1 \circ \cdots \circ \tilde{\sigma}_n) \\ &= ((\alpha_b, \beta_b)\tilde{\sigma}_1 \circ \cdots \circ (\alpha_b, \beta_b)\tilde{\sigma}_n) \\ \text{st } T &= \dfrac{T_i}{\beta_b} \end{cases}$$

And with that we have finished re-defining the MDLe specification so that we can temporally compress or dilate any signal to a new time frame, and use vector addition between the atoms in our alphabet. These definitions will be assumed to be default for the rest of this thesis.

3.2 Building and Using the Atom Alphabet Σ_a

With this new version of MDLe, we have gained the ability to temporally scale a signal so that it can exist within the $L_2[0, T]$ subspace spanned by our atoms in Σ_a. And once in this time frame, we have gained the ability to use vector addition with our atoms through the merge operator, which means we wield the power of the projection theorem.

In this section, we will use this specification to finally build a theory in which we can build an atom alphabet Σ_a from any given control signal. This will be followed

by a theory allowing us to use an existing Σ_a to reconstruct any signal based only on the atoms within Σ_a.

3.2.1 Building the Atom Alphabet

Before we re-specified MDLe, we had developed a method of segmenting a given control signal into a sequential concatenation of n signals. So we return to our original question, how can we build an alphabet Σ_a from these n signals?

As discussed earlier, we want the atoms in Σ_a to be a set of basis vectors in an $L_2[0,T]$ subspace. Conceptually, these atoms should have the capacity to replicate each of the n signal segments through the use of the merge operator. This translates to making sure that each of the n signals lies in the space spanned by Σ_a.

We can check this by first projecting the signal into the space defined by Σ_a, resulting in a merged combination of atoms in Σ_a. We can then check to see if the resulting MDLe signal is the same as the original. If so, then the alphabet is indeed rich enough to recreate this segment of the original signal.

So how do we do we project these signal segments onto our $L_2[0,T]$ alphabet space?

3.2.1.1 Projections onto our Alphabet

For simplicity, from now on we will denote any scaled controls $u_i(\beta_i t)$ that exists in our $L_2[0,T]$ alphabet space as u_i.

To begin, suppose we have an alphabet $\Sigma_a = \{\sigma_1, \ldots, \sigma_n\}$, and are given a signal segment u_r we want to project onto our alphabet. This signal can be defined as the combination of two signals that lie in the space Σ_a, and the orthogonal space $\Sigma_a^{\perp}$. We can write this as

$$u_r = u^* + u^{*\perp} \quad st\ u^* \in \Sigma_a,\ u^{*\perp} \notin \Sigma_a$$

Here u^* is in the span of our alphabet, and is thus the projection of u_r onto Σ_a. So the task of projecting u^* onto Σ_a comes down to finding the set of scalars $\{\alpha_1, \ldots \alpha_n\}$

such that

$$u^* = \alpha_1 u_1 + \cdots + \alpha_n u_n$$

where u^* solves the least squares criterion.

$$||u_r - u^*|| \leq ||u_r - u|| \quad \forall u \in \Sigma_a \tag{2}$$

We use this criterion because it is guaranteed to create the closest approximation of u_r we can make using our alphabet Σ_a.

Well we know that this unique minimizing vector u^* is the orthogonal projection of u_r onto Σ_a, which is the combination of the projections of u_r onto each basis vector in Σ_a. So let's project u_r onto the first basis vector u_1.

$$\begin{aligned}
\langle u_r, u_1 \rangle &= \langle u^*, u_1 \rangle + \cancelto{0}{\langle u^{*\perp}, u_1 \rangle} \\
\langle u_r, u_1 \rangle &= \langle \alpha_1 u_1 + \cdots + \alpha_n u_n, u_1 \rangle \\
\langle u_r, u_1 \rangle &= \langle \alpha_1 u_1, u_1 \rangle + \cdots + \langle \alpha_n u_n, u_1 \rangle \\
\langle u_r, u_1 \rangle &= \alpha_1 \langle u_1, u_1 \rangle + \cdots + \alpha_n \langle u_n, u_1 \rangle
\end{aligned} \tag{3}$$

We then extend this process to every basis vector $u_i \in \Sigma_a$.

$$\begin{aligned}
\langle u_r, u_1 \rangle &= \alpha_1 \langle u_1, u_1 \rangle + \cdots + \alpha_n \langle u_n, u_1 \rangle \\
&\vdots \qquad\qquad\qquad \vdots \\
\langle u_r, u_n \rangle &= \alpha_1 \langle u_1, u_n \rangle + \cdots + \alpha_n \langle u_n, u_n \rangle
\end{aligned} \tag{4}$$

And we see that this process is a linear transformation of the α_i scalars we are looking for.

$$\begin{bmatrix} \langle u_r, u_1 \rangle \\ \vdots \\ \langle u_r, u_n \rangle \end{bmatrix} = \begin{bmatrix} \langle u_1, u_1 \rangle & \cdots & \langle u_n, u_1 \rangle \\ \vdots & \ddots & \vdots \\ \langle u_1, u_n \rangle & \cdots & \langle u_n, u_n \rangle \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix} \tag{5}$$

This equation can be rewritten as $v = G\alpha$, where G is classically known as the *Gram Matrix*. And as we know from [9], the *Gram determinant* $g \neq 0$ if and only if the vectors $u_1, \ldots, u_n$ are linearly independent. And since by the definition of Σ_a we

know that $u_1, \ldots, u_n$ are linearly independent, G is always invertible. This leads us to a solution for $\alpha_1, \ldots \alpha_n$, and thus the projection of u_r onto Σ_a.

Theorem 3.2.1. *Spatial Projection of u_r onto Σ_a:*

$$\textit{Given } u_r = u^* + u^{*\perp} \quad \textit{st } u^* \in \Sigma_a,\ u^{*\perp} \notin \Sigma_a$$

$$\begin{cases} u^* & = & proj_{\Sigma_a}(u_r) \\ proj_{\Sigma_a}(u_r) & = & \alpha^* \Sigma_a \\ \alpha^* & = & G^{-1} v \end{cases}$$

$$\textit{where } \alpha^* \in \mathbb{R}^n,\ G \in \mathbb{R}^{n \times n} \textit{ is the Gram matrix}, v \in \mathbb{R}^n$$

3.2.1.2 Recognizing Controls Orthogonal to the Alphabet

So now that we can create the spatial projections of any signal u_r onto the alphabet Σ_a, how can we tell whether or not the projection is an exact copy of the original signal?

We know from the previous section that we can write

$$u_r = u^* + u^\perp \quad st\ u \in \Sigma_a,\ u^\perp \notin \Sigma_a$$

where $u^* = proj_{\Sigma_a}(u_r)$. We also know that the only time the $proj_{\Sigma_a}(u_r) = u_r$ is when the orthogonal component $u^\perp = 0$. Therefore a simple test of whether our alphabet is rich enough to recreate u_r is to make sure that $u_r - proj_{\Sigma_a}(u_r) = 0,\ \forall t \in [0, T]$. Unfortunately this requires a comparison for every point in time. Instead, it would be nice if we could have a single value to tell us whether or not our alphabet is rich enough. We can do this by examining the magnitude of u_r.

$$||u_r||^2 = ||u^* + u^\perp||^2$$

This can be rewritten as

$$\begin{aligned} ||u_r||^2 &= \langle u^* + u^\perp, u^* + u^\perp \rangle \\ ||u_r||^2 &= \langle u^*, u^* \rangle + \langle u^\perp, u^\perp \rangle + 2\langle u^*, u^\perp \rangle \end{aligned}$$

And since we know that u^* and $u^\perp$ are orthogonal, ie $\langle u^*, u^\perp \rangle = 0$, then

$$||u_r||^2 = \langle u^*, u^* \rangle + ||u^\perp||^2$$

We can further simplify this by expanding out $\langle u^*, u^* \rangle$

$$\begin{aligned}
\langle u^*, u^* \rangle &= \langle (\alpha_1 u_1 + \cdots + \alpha_n u_n), (\alpha_1 u_1 + \cdots + \alpha_n u_n) \rangle \\
&= \alpha_1(\langle u_1, \alpha_1 u_1 \rangle + \cdots + \langle u_1, \alpha_n u_n \rangle) + \cdots + \alpha_n(\langle u_1, \alpha_1 u_1 \rangle + \cdots + \langle u_1, \alpha_n u_n \rangle) \\
&= \begin{bmatrix} \alpha_1, \ldots, \alpha_n \end{bmatrix} \begin{bmatrix} \alpha_1 \langle u_1, u_1 \rangle + \cdots + \alpha_n \langle u_n, u_1 \rangle \\ \vdots \\ \alpha_1 \langle u_1, u_n \rangle + \cdots + \alpha_n \langle u_n, u_n \rangle \end{bmatrix} \\
&= \begin{bmatrix} \alpha_1, \ldots, \alpha_n \end{bmatrix} \begin{bmatrix} \langle u_1, u_1 \rangle & \cdots & \langle u_n, u_1 \rangle \\ \vdots & \ddots & \vdots \\ \langle u_1, u_n \rangle & \cdots & \langle u_n, u_n \rangle \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_n \end{bmatrix}
\end{aligned}$$

And so we recognize this as a combination of our alpha and Gram matrices.

$$||u_r||^2 = \alpha^T G \alpha + ||u^\perp||^2 \tag{6}$$

This leads us to a theorem for determining whether our alphabet Σ_a is rich enough to replicate a signal segment u_r.

Theorem 3.2.2. *Given a signal u_r, we can replicate u_r using Σ_a if:*

$$||u_r||^2 - \alpha^T G \alpha = 0$$

where $\alpha = G^{-1} v$, and G is the Gram matrix

3.2.1.3 Algorithm to Build the Atom Alphabet

Having defined a theorem that allows us to project signal segments onto our alphabet 3.2.1, as well as a test to determine if the signal can be perfectly recreated using a combination of atoms in our alphabet 3.2.2, we are finally ready to develop a

theoretical algorithm that can be used to build an atom alphabet from a given control signal.

Assume we are given a control signal $u_r(t)$ from which we wish to build or augment an alphabet Σ_a. And assume this Σ_a corresponds to an $L_2[0,T]$ space with the signals $\{u_{\sigma_1}, \ldots, u_{\sigma_n}\}$ acting as basis vectors. Our first step is to temporally segment $u_r(t)$ into m distinct signal segments as defined in section 3.1.1. We then temporally scale all of the segmented signals $\{u_1, \ldots, u_m\}$ such that they exist on the time period $t \in [0,T]$.

The main purpose in building the alphabet Σ_a is to develop enough basis vectors such that every segmented signal $\{u_1, \ldots, u_m\}$ we have identified exists in the final space Σ_a, allowing us to completely replicate each signal segment, and thus the entire original signal $u_r(t)$.

So we must look at every segmented signal individually to determine if it is in the space Σ_a. If it is in the space, then no action is needed because we have the ability to replicate this particular signal. But if it is not in the space, we will need to develop a new atom σ_{n+1} such that the signal now lies in the augmented space $\Sigma_a \oplus \sigma_{n+1}$.

So for each signal in $\{u_1, \ldots, u_m\}$, we first project u_i onto Σ_a as defined in theorem (3.2.1).

$$u^* = proj_{\Sigma_a}(u_i)$$

We then run the test described in theorem (3.2.2) to determine whether or not $u^* \in \Sigma_a$. If it is in the space, then we move on to analyze the next signal. But if it does not lie in Σ_a, we must find a new signal to add to Σ_a.

As we know

$$u_i = proj_{\Sigma_a}(u_i) + u_i^{\perp}$$

Obviously, the only component we can not make is $u_i^{\perp}$. So let's add this component into our alphabet.

$$\Sigma_a = \Sigma_a \oplus \sigma_{n+1} \quad st\ \sigma_{n+1} = (u_i - proj_{\Sigma_a}(u_i), \xi, T) \tag{7}$$

```
INPUT: u_r(t); Σ_a;
begin
    segment(u_r) → {u_1(t), ..., u_m(t)};
    scale (u_i) → {u_1, ..., u_m};
    compute Gram matrix G;
    while i ≤ m do
        compute v as defined in 3.2.1;
        compute α = G^{-1} v as defined in 3.2.1;
        compute ||u_i||^2;
        if ||u_i||^2 − α^T G α ≠ 0 then
            compute u_i^⊥ as defined in (7);
            augment alphabet Σ_a = Σ_a ⊕ σ_{n+1};
            recompute G;
        end
        i = i + 1;
    end
end
OUTPUT: Σ_a
```

Figure 5: Theoretical Algorithm for Atom Alphabet Creation

Now by eliminating the only part of u_i that we could not mimic with the old Σ_a, we can re-project u_i onto the new Σ_a, where there will now exist a vector α st theorem (3.2.2) holds true.

So in using this outline for atom alphabet creation, we are able to create an alphabet that can mimic all reference trajectories it has been asked to mimic. And as an added benefit, all atoms in the alphabet will exist in the same time period $t \in [0, T]$, which will greatly simplify its use, as will be shown in the next section.

To conclude this section on atom alphabet creation, I leave the reader with a detailed algorithm, shown in figure 5.

3.2.2 Using the Atom Alphabet

Suppose we run the method defined in section 3.2.1.3 with a large data set of signals gathered from high-level users using a robot to complete various tasks. By our expectations, this Σ_a will contain the ability to replicate any new signals generated by

a high-level user using the robot for similar tasks. Thus we should develop a method to decompose a new signal $u_r(t)$ into an MDLe string based on the atoms in Σ_a that will replicate, or closely approximate, $u_r(t)$. In other words:

$$u_r(t) = \tilde{\sigma}_1 \circ \cdots \circ \tilde{\sigma}_m$$

where m is the number of temporal signal segments found in $u_r(t)$, and $\tilde{\sigma}_i$ is a merged atom as defined in equation 3.1.3.

The task of decomposing $u_r(t)$ into this MDLe string comes down to segmenting the signal into a group of m distinct segments, and then finding the (α_i, β_i) scalars inside of each merged atom that will recreate the original signal.

To begin, let us again use the method of section 3.1.1 to segment $u_r(t)$ into m distinct segments such that

$$\begin{aligned}
u_r(t) &= u_1(t) && 0 \leq t < \tau_1 \\
u_r(t) &= u_2(t - \tau_1) && \tau_1 \leq t < \tau_2 \\
&\vdots \\
u_r(t) &= u_m(t - \tau_{m-1}) && \tau_{m-1} \leq t < \tau_m
\end{aligned}$$

We can then find a vector $T_s = [T_1, T_2, \ldots, T_m]$ such that $T_i = \tau_i - \tau_{i-1}$ and is the time period for which each signal segment is defined on. We can illustrate how this relates to our future MDLe call in a form like the following

$$u_r(t) = (u_1, \xi, T_1) \circ (u_2, \xi, T_2) \circ \cdots \circ (u_m, \xi, T_m)$$

Now that we have a template for our MDLe call, we need to scale each segment into the $[0, T]$ time frame so we can project them onto the alphabet Σ_a. So let us define a vector $\bar{\beta} = [\bar{\beta}_1, \ldots, \bar{\beta}_m]$ such that $\bar{\beta}_i = \dfrac{T_i}{T}$ is the temporal scalar putting each signal segment into the time frame $t \in [0, T]$. We can also define a dual vector containing the inverse scalars $\beta = [\frac{1}{\bar{\beta}_1}, \ldots, \frac{1}{\bar{\beta}_1}]$ such that $\beta_i = \dfrac{T}{T_i}$. These β scalars allow us to re-scale the signals from the Σ_a time frame $[0, T]$, back to their original period $[0, T_i]$.

We can now define a relationship between each original segment $u_i(t)\ , t \in [0, T_i]$ and its scaled segment $\tilde{u}_i(t)\ , t \in [0, T]$ we need to work in the Σ_a space.

$$\begin{cases} \tilde{u}_i(t) &= u_i(\bar{\beta}_i t) &\rightarrow (u_i(\bar{\beta}_i t), \xi, \dfrac{T_i}{\bar{\beta}_i} = T)\ \ , \ t \in [0, T] \\ & &\rightarrow (1, \bar{\beta}_i)(u_i, \xi, T_i) \\ u_i(t) &= \tilde{u}_i(\beta_i t) &\rightarrow (\tilde{u}_i(\beta_i t), \xi, \dfrac{T}{\beta_i} = T_i)\ , \ t \in [0, T_i] \\ & &\rightarrow (1, \beta_i)(\tilde{u}_i, \xi, T) \end{cases} \tag{8}$$

These definitions allow us to redefine our original signal $u_r(t)$ using MDLe calls based on the temporally scaled segments $\{\tilde{u}_1, \ldots, \tilde{u}_m\}$.

$$\begin{cases} u_r(t) &= (1,1)u_1 \circ \cdots \circ (1,1)u_m \\ &= (1, \beta_1)(\tilde{u}_1, \xi, T) \circ \cdots \circ (1, \beta_m)(\tilde{u}_m, \xi, T) \end{cases} \tag{9}$$

And so we see that if we can find the signals $\{\tilde{u}_1, \ldots, \tilde{u}_m\}$ in (9) that are based on atoms from Σ_a, we just need to apply the β_i scalar and that segment will retake its rightful place in the original signal $u_r(t)$.

So to find these values $\tilde{u}_i$ as a parallel combination of atoms in Σ_a, we can project each signal onto the alphabet as described in theorem (3.2.1), resulting in an MDLe merged atom.

$$\begin{cases} \tilde{u}_i^* &= proj_{\Sigma_a}(\tilde{u}_i) \\ &= \alpha_1 u_{\sigma_1} + \cdots + \alpha_n u_{\sigma_n} \\ &= (\alpha_1, 1)\sigma_1 || \cdots || (\alpha_n, 1)\sigma_n \end{cases} \tag{10}$$

We can then insert this back into (9) to get

$$u_r(t) = (1, \beta_1)(proj_{\Sigma_a}(\tilde{u}_1), \xi, T) \circ \cdots \circ (1, \beta_m)(proj_{\Sigma_a}(\tilde{u}_m), \xi, T)$$

Which we can expand into

$$\begin{aligned} u_r(t) &= (1, \beta_1)(\alpha_{11} u_{\sigma_1} + \cdots + \alpha_{1n} u_{\sigma_n}, \xi, T) \circ \cdots \\ &\quad \cdots \circ (1, \beta_m)(\alpha_{m1} u_{\sigma_1} + \cdots + \alpha_{mn} u_{\sigma_n}, \xi, T) \end{aligned}$$

And further into

$$\begin{aligned} u_r(t) &= (\alpha_{11},\beta_1)(u_{\sigma_1},\xi,T)||\cdots||(\alpha_{1n},\beta_1)(u_{\sigma_n},\xi,T)\circ\cdots \\ &\quad \cdots\circ(\alpha_{m1},\beta_m)(u_{\sigma_1},\xi,T)||\cdots||(\alpha_{mn},\beta_m)(u_{\sigma_n},\xi,T) \end{aligned}$$

And finally, into the MDLe call

$$\begin{aligned} u_r(t) &= (\alpha_{11},\beta_1)\sigma_1||\cdots||(\alpha_{1n},\beta_1)\sigma_n\circ\cdots \\ &\quad \cdots\circ(\alpha_{m1},\beta_m)\sigma_1||\cdots||(\alpha_{mn},\beta_m)\sigma_n \end{aligned}$$

And thus we arrive at a formal definition for the decomposition of a signal $u_r(t)$ into an MDLe atom call based on Σ_a.

Theorom Given an atom alphabet Σ_a and a signal u_r segmented into a temporary MDLe call of $u_r(t) = \sigma_{r1}\circ\cdots\circ\sigma_{rm}$, we can decompose u_r into an MDLe call of the following form:

$$u_r(t) = \tilde{\sigma}_1\circ\cdots\circ\tilde{\sigma}_m$$

$$\text{where } \begin{cases} \tilde{\sigma}_i = (\alpha_{i1},\beta_i)\sigma_1||\cdots||(\alpha_{in},\beta_i)\sigma_n \\ \alpha_i = [\alpha_{i1},\ldots,\alpha_{i,n}] = proj_{\Sigma_a}((1,\bar{\beta}_i)\sigma_{ri}) \\ \beta_i = \dfrac{T}{T_i}\ , \bar{\beta}_i = \dfrac{T_i}{T} \end{cases} \tag{11}$$

This MDLe call will create a signal that is the best approximation we can make based off of the current alphabet Σ_a. And in the case that every signal segment identified was in the space Σ_a, then the copy will be an exact match.

We conclude this section with a detailed algorithm for decomposing $u_r(t)$ into an MDLe call.

3.3 Building and Using the Behavioral Alphabet Σ_b

At this point, we have successfully developed a theoretical algorithm that can build and use an atom alphabet through the method of bottom-up mimicry. And by our hypothesis of how bottom-up mimicry should work, this Σ_a could be trained with

INPUT: $u_r(t)$; Σ_a;
begin
  segment$(u_r) \rightarrow \sigma_{r1} \circ \cdots \circ \sigma_{rm}$;
  scale $(\sigma_{ri}) \rightarrow (1, \bar{\beta}_i)(u_{ri}, \xi, T_i)$;
  compute *Gram matrix* G;
  while $i \leq m$ **do**
    compute v from $(1, \bar{\beta}_i)\sigma_{ri}$ as defined in (3.2.1);
    compute $\alpha = G^{-1}v$ as defined in (3.2.1);
    $MDLe_i = (\alpha, \beta_i)\Sigma_a$;
    $MDLe_a = MDLe_a \circ MDLe_i$;
    $i = i + 1$;
  end
end
OUTPUT: $MDLe_a$

Figure 6: Theoretical Algorithm for Decomposing a Control into an MDLe Atom String

various data sets to harness the ability to create the signals required to run various high-level tasks.

But as you may have noticed in section 3.2.1.3 when we built Σ_a, the atoms are defined in a pretty arbitrary manner. For one, their temporal extent is limited to the boundaries we defined using the theory of section 3.1.1. So if a high level task straddles a temporal border, there is no chance that it is defined as an atom, and thus can only be represented by a sequence of merged atom calls. If this task is commonly used, we will have to continuously keep calling these merged atom strings to recreate the task, rather than a single call.

Secondly, the atoms added to Σ_a were only the orthogonal components of some signal to the alphabet itself. This means that the atoms in Σ_a are only interested in spanning a space, and may never appear as itself in the original signal $u_r(t)$, only as a component of a merged atom. And so if we have a signal corresponding to a commonly used task which can only be created using a specific merged atom call, we would prefer to define this merged atom and call it by name.

These issues can be overcome by utilizing the higher-level definitions of MDLe:

behaviors. So in this section we will develop a method for finding and using commonly occurring behaviors that straddle our original temporal boundaries, and subsume the sets of merged atoms. In this respect, it is possible that these behaviors will more closely represent the signals required for commonly performed, high-level tasks.

3.3.1 Identifying a Behavior

Suppose we are given an MDLe$_a$ call such as the following.

$$u_r(t) = \cdots \circ \tilde{\sigma}_1 \circ \tilde{\sigma}_2 \circ \cdots \circ (\alpha_b, \beta_b)\tilde{\sigma}_1 \circ (\alpha_b, \beta_b)\tilde{\sigma}_2 \circ \cdots$$

In examining this sequence we see that the string of merged atoms $\tilde{\sigma}_1 \circ \tilde{\sigma}_2$ occurs twice, except in the second call both merged atoms are scaled by the factor (α_b, β_b).

So if this exact signal occurs more than once within an MDLe$_a$ call, we can make the assumption that it is a common task being performed by the higher level user. So let us define it as a behavior as follows.

$$\pi_1 = (\tilde{\sigma}_1 \circ \tilde{\sigma}_2)$$

With this definition, we can now form a behavioral MDLe (MDLe$_b$) string for $u_r(t)$ that will produce the same signal as the MDLe$_a$ string.

$$u_r(t) = \cdots \circ \pi_1 \circ \cdots \circ (\alpha_b, \beta_b)\pi_1 \circ \cdots$$

This concept is what we mean when we say we are looking for common behaviors within an MDLe$_a$ call.

3.3.1.1 Defining a Behavior

Suppose we are given the MDLe$_a$ string

$$u_r(t) = \cdots \circ \tilde{\sigma}_1 \circ \tilde{\sigma}_2 \circ \cdots \circ (\alpha_b, \beta_b)\tilde{\sigma}_1 \circ (\alpha_b, \beta_b)\tilde{\sigma}_2 \circ \cdots$$

Upon examination it is easy to see that a behavior exists. But we must remember that the definition of a merged atom $\tilde{\sigma}_i$ is for notation only, and that the actual call

looks more like this.

$$\cdots \circ (\alpha_{11}, \beta_1)\sigma_1 || \cdots || (\alpha_{1n}, \beta_1)\sigma_n \circ (\alpha_{21}, \beta_2)\sigma_1 || \cdots || (\alpha_{2n}, \beta_2)\sigma_n \circ \cdots$$
$$\cdots \circ (\alpha_b\alpha_{11}, \beta_b\beta_1)\sigma_1 || \cdots || (\alpha_b\alpha_{1n}, \beta_b\beta_1)\sigma_n \circ (\alpha_b\alpha_{21}, \beta_b\beta_2)\sigma_1 || \cdots || (\alpha_b\alpha_{2n}, \beta_b\beta_2)\sigma_n \circ \cdots$$

Further, in this example we have singled out these calls from a string. But when searching from scratch for these two sets, they will be buried in the midst of a wide variety of calls.

But the biggest issue we face when defining behaviors is that we didn't even know that we were looking for these two sets of merged atoms. When blindly looking for behaviors we have no notion of which sets of merged atoms might be part of a behavior, how long the behavior is, or where it begins and ends. It is tantamount to looking for a needle in a haystack, except you aren't exactly sure what the needle looks like.

So it should be apparent to the reader that this is a conceptually difficult task. But it can made easier if we adopt a new notation for a merged atom.

To begin, lets take a look at how the merged atoms in our MDLe$_a$ calls will appear when using the method defined in section 3.2.2.

$$\tilde{\sigma}_i = (\alpha_{i1}, \beta_i)\sigma_1 || \cdots || (\alpha_{in}, \beta_i)\sigma_n$$

We see that every merged atom is a combination of the basis vectors in Σ_a, a set of α scalars, and a common β scalar due to the fact that all of the signals in Σ_a exist in the same time scale $t \in [0, T]$. So lets create an alternate notation for these versions of $\tilde{\sigma}_i$.

$$\begin{cases} \tilde{\sigma}_i = (\alpha_i, \beta_i)\Sigma_a \\ \text{where } \alpha_i = [\alpha_{i1}, \ldots, \alpha_{in}] \end{cases} \tag{12}$$

Let us now apply this definition to the string of merged atoms $s_r = \tilde{\sigma}_1 \circ \cdots \circ \tilde{\sigma}_m$.

$$s_r = (\alpha_1, \beta_1)\Sigma_a \circ \cdots \circ (\alpha_m, \beta_m)\Sigma_a$$

Now, since all of the calls rely on Σ_a, we can remove it from the equation, knowing that we haven't lost any information we can't recover.

$$s_r = (\alpha_1, \beta_1) \circ \cdots \circ (\alpha_m, \beta_m)$$

We see that the string s_r can be represented by a string of 2-tuples containing the important information on what differentiates each merged atom call from the others. We can simplify this one more time by ordering the sequential calls into two matrices, in which each subsequent row vector corresponds to a subsequent MDLe$_a$ call.

Definition 3.3.1. MDLe$_a$ string in 2-tuple form st. $s_r = (A, B)$

$$s_r = \left(\begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix}, \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \right)$$

Here, A is a matrix containing the α_i vectors, B is a vector containing all of the β_i scalars, and each descending row vector represents a subsequent merged atom call in s_r.

This notation turns out to be very useful in finding a behavior. To show how, let us now declare two strings $s_i = \tilde{\sigma}_1 \circ \cdots \circ \tilde{\sigma}_m$, $s_j = (\alpha_b, \beta_b)\tilde{\sigma}_1 \circ \cdots \circ (\alpha_b, \beta_b)\tilde{\sigma}_m$ that are the same behavior, but scaled differently. If we use the notation for a scaled atom defined in (12), our two strings look as follows.

$$\begin{cases} s_i = (\alpha_1, \beta_1)\Sigma_a \circ \cdots \circ (\alpha_m, \beta_m)\Sigma_a \\ s_j = (\alpha_b\alpha_1, \beta_b\beta_1)\Sigma_a \circ \cdots \circ (\alpha_b\alpha_m, \beta_b\beta_m)\Sigma_a \end{cases}$$

And putting this into the 2-tuple form of definition 3.3.1.

$$s_i = \left(\begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix}, \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \right) \quad s_j = \left(\begin{bmatrix} \alpha_b\alpha_1 \\ \vdots \\ \alpha_b\alpha_m \end{bmatrix}, \begin{bmatrix} \beta_b\beta_1 \\ \vdots \\ \beta_b\beta_m \end{bmatrix} \right)$$

We can extract the scalars in s_j, giving us the following form.

$$s_i = \left(\begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix}, \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \right) \quad s_j = \left(\alpha_b \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix}, \beta_b \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \right)$$

And here is where we see a solution to our problems. For this case to validly exist, two conditions must hold true. First, each row vector in A and B from the tuples of the string s_j have to be linearly dependent to the corresponding row vectors in A and B from s_i. Second, each scalar relating the linear dependence of the rows in s_j to the rows in s_i have to be the same, and correspond to (α_b, β_b).

And so we can put a formal definition on when something is a behavior.

Theorem 3.3.2. *Given two strings of merged atoms $s_i = \tilde{\sigma}_{i1} \circ \cdots \circ \tilde{\sigma}_{im}$, $s_j = \tilde{\sigma}_{j1} \circ \cdots \circ \tilde{\sigma}_{jm}$ in 2-tuple form, with $m \geq 1$, $\exists$ a behavior if:*

(i) Each corresponding row vector (α_k, β_k) in s_i,s_j is linearly dependent for $k \in [1, m]$

(ii) Each set of scalars (x_k, y_k) such that $\alpha_{ik} = x_k \cdot \alpha_{jk}$, $\beta_{ik} = y_k \cdot \beta_{jk}$ is equal $\forall k \in [1, m]$.

If (i) and (ii) hold, then $\pi = s_i$, $(\alpha_b, \beta_b) = (x_k, y_k)\ \forall k \in [1, m]$, and $s_j = (\alpha_b, \beta_b)\pi$.

3.3.1.2 Finding a Possible Behavior

Now that we have a condition for how to identify a valid behavior, we can use it to find these behaviors in an MDLe$_a$ call.

Suppose we are given an MDLe$_a$ string that contains an undefined behavior.

$$\tilde{\sigma}_1 \circ \cdots \circ \tilde{\sigma}_m$$

Using the notation of definition 3.3.1, we can reorder this call as a 2-tuple of the

following form.

$$\left(\begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix}, \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_m \end{bmatrix} \right)$$

We know from theorem 3.3.2 that a valid behavior can be defined by two strings that have the same sequence of linearly dependent row vectors. So if a behavior exists in this string, there exists two separate sub 2-tuples that are linearly dependent. ie.

$$\left(\begin{bmatrix} \vdots \\ \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_k \end{bmatrix} \\ \vdots \\ \alpha_b \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_k \end{bmatrix} \\ \vdots \end{bmatrix}, \begin{bmatrix} \vdots \\ \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_k \end{bmatrix} \\ \vdots \\ \beta_b \begin{bmatrix} \beta_1 \\ \vdots \\ \beta_k \end{bmatrix} \\ \vdots \end{bmatrix} \right)$$

So lets create a function in which we can find these strings of linearly dependent row vectors quickly. We can do this by defining states to each row vector in (A, B), where two row vectors only share the same state if they are linearly dependent.

Definition 3.3.3. A state is an integer $z_i \in \mathbb{Z}^+$ corresponding to each row vector α in a matrix A defined as follows:

(i) $z_i = z_j$ iff $\exists k$ st $\alpha_i = k\alpha_j$, $\forall i, j \in [1, m]$

(ii) $z_i = 1$ iff $\alpha_i = 0$, $\forall i \in [1, m]$

Thus the previous example containing the sequence of linearly dependent vectors

might correspond to a state 2-tuple of the following form.

$$\left(\begin{bmatrix} \vdots \\ \begin{bmatrix} 3 \\ 5 \\ 8 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 3 \\ 5 \\ 8 \end{bmatrix} \\ \vdots \end{bmatrix}, \begin{bmatrix} \vdots \\ \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \\ \vdots \\ \begin{bmatrix} 2 \\ 2 \\ 2 \end{bmatrix} \\ \vdots \end{bmatrix} \right)$$

These common state strings are indicative of a potential behavior, as they represent two similar sequences of linearly dependent vectors. Thus we define the phenomenon as a common subsequence.

Definition 3.3.4. A **common subsequence** of a matrix A is a string of states $\bar{z} = (z_1, \ldots, z_n)$, $n \geq 1$ that occurs at least twice in a state vector.

In relation to our MDLe$_a$ 2-tuple (A,B), suppose we find a common subsequence in A represented by the state strings $\bar{z}_{i_\alpha}, \bar{z}_{j_\alpha}$ such that $\bar{z}_{i_\alpha} = \bar{z}_{j_\alpha}$. If the corresponding strings in B $\bar{z}_{i_\beta}, \bar{z}_{j_\beta}$ are also equal, then the first condition of linearly dependent row vectors of theorem 3.3.2 has been satisfied. This gives us a corollary to theorem 3.3.2.

Corollary 3.3.5. *Given two strings of merged atoms s_i, s_j in $s_r = (A, B)$ with indexes in $(A, B) = \{[s_{i_{start}}, s_{i_{end}}], [s_{j_{start}}, s_{j_{end}}]\}$, and a corresponding state 2-tuple $Z_r = (A_{\bar{z}}, B_{\bar{z}})$; s_i, s_j satisfy condition (i) of theorem 3.3.2 if:*

(i) $A_{\bar{z}}(s_{i_{start}} : s_{i_{end}}, :) = A_{\bar{z}}(s_{j_{start}} : s_{j_{end}}, :)$

(ii) $B_{\bar{z}}(s_{i_{start}} : s_{i_{end}}, :) = B_{\bar{z}}(s_{j_{start}} : s_{j_{end}}, :)$

So we now have an easy means of identifying common subsequences with a guarantee that they satisfy half of the conditions needed to define a valid behavior. But how do we use this to satisfy the second condition of theorem 3.3.2 in which we need to verify that the scalars relating all of the linearly dependent vectors are equal?

Well suppose we pull out the sub-matrices of A from the MDLe$_a$ 2-tuple corresponding to the locations of a common subsequence we've found. They and their corresponding state vectors will look as follows.

$$s_i = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix} \rightarrow \bar{z}_i = \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix} \quad , \quad s_j = \begin{bmatrix} \alpha_{b_1}\alpha_1 \\ \vdots \\ \alpha_{b_m}\alpha_m \end{bmatrix} \rightarrow \bar{z}_j = \begin{bmatrix} z_1 \\ \vdots \\ z_m \end{bmatrix}$$

We see that in the action of defining the state vectors, we've lost the scaling information we need to verify condition (ii) of theorem 3.3.2. So let us find a process that retains this scaling information as we define the states.

Assume we have a set of vectors $\{\alpha_1, \ldots, \alpha_m\}$ that share the same state, ie $z_i = \cdots = z_m$. Let us call the vector α_1 of the set the base vector. We can then find a vector k of the form

$$k = [proj_{\alpha_1}(\alpha_1), proj_{\alpha_1}(\alpha_2), \ldots, proj_{\alpha_1}(\alpha_m)]$$

which consists of the scalars relating the linear dependencies of each vector to the base vector.

Now if we create a vector k_i of this form for each unique state z_i found in a matrix A, we have a means of retaining the scaling relationship between any two linearly dependent vectors in the set defined by the base vector α_1. A process of this form would work on the previous example as follows, assuming the vectors in s_i are defined as base vectors.

$$s_i = \begin{bmatrix} \alpha_1 \\ \vdots \\ \alpha_m \end{bmatrix} \rightarrow \bar{k}_i = \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \quad , \quad s_j = \begin{bmatrix} \alpha_{b_1}\alpha_1 \\ \vdots \\ \alpha_{b_m}\alpha_m \end{bmatrix} \rightarrow \bar{k}_j = \begin{bmatrix} \alpha_{b_1} \\ \vdots \\ \alpha_{b_m} \end{bmatrix}$$

So if we combine the state finding process with this scalar finding process, the result will be a 2-tuple containing enough information to determine if two substrings satisfy both conditions of theorem 3.3.2, and can thus be considered a valid behavior.

Definition 3.3.6. A process $\mathbf{f(A)} = (A_{\bar{z}}, A_{\bar{k}})$ of any given matrix A:

(i) $A_{\bar{z}}$ is a vector containing the states of each row vector in A as defined in definition 3.3.3.

(ii) $A_{\bar{k}}$: given m unique states in $A_{\bar{z}}$, for each state $i \in [1, m]$ define a set $\{\alpha_1, \ldots, \alpha_n\}$ containing the n vectors of A corresponding to state z_i, with an inverse mapping back to their proper index. Let

$$k_i = [proj_{\alpha_1}(\alpha_1), proj_{\alpha_1}(\alpha_2), \ldots, proj_{\alpha_1}(\alpha_n)]$$

$A_{\bar{k}}$ contains the values of k_i, $\forall i \in [1, m]$, inversely mapped to their proper indexes.

This process is defined for a single matrix A. We will conclude this section by defining a parallel process for our MDLe$_a$ 2-tuples which we will use in the next section to identify valid behaviors.

Definition 3.3.7. A process $\mathbf{g((A,B))} = ((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$ of any given 2-tuple (A, B):

$$g((A, B)) = (f(A), f(B))$$

3.3.1.3 Finding a Valid Behavior

The process defined in definition 3.3.7 is the real key to finding potential behaviors. In this section I will show why.

Assume we are given an MDLe$_a$ 2-tuple (A, B) containing an undefined behavior. We can run the process $g(A, B)$ on it to obtain the 2-tuple $((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$. Now

suppose we identify a common subsequence in $A_{\bar{z}}$ and $B_{\bar{z}}$ that satisfies corollary 3.3.5. This means that we have identified multiple occurrences of a sequence of row vectors such that in each occurrence, the row vectors are linearly dependent. The submatrices of $((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$ corresponding to these occurrences will look like this.

$$s_i = \left(\left(\begin{bmatrix} z_{1_\alpha} \\ \vdots \\ z_{m_\alpha} \end{bmatrix}, \begin{bmatrix} k_{i1_\alpha} \\ \vdots \\ k_{im_\alpha} \end{bmatrix} \right), \left(\begin{bmatrix} z_{1_\beta} \\ \vdots \\ z_{m_\beta} \end{bmatrix}, \begin{bmatrix} k_{i1_\beta} \\ \vdots \\ k_{im_\beta} \end{bmatrix} \right) \right)$$

$$s_j = \left(\left(\begin{bmatrix} z_{1_\alpha} \\ \vdots \\ z_{m_\alpha} \end{bmatrix}, \begin{bmatrix} k_{j1_\alpha} \\ \vdots \\ k_{jm_\alpha} \end{bmatrix} \right), \left(\begin{bmatrix} z_{1_\beta} \\ \vdots \\ z_{m_\beta} \end{bmatrix}, \begin{bmatrix} k_{j1_\beta} \\ \vdots \\ k_{jm_\beta} \end{bmatrix} \right) \right)$$

We can normalize everything with respect to s_i if $\forall p \in [1, m],\ k_{ip_\alpha} = \alpha_{bp} \cdot k_{jp_\alpha}$. This gives us

$$s_i = \left(\left(\begin{bmatrix} z_{1_\alpha} \\ \vdots \\ z_{m_\alpha} \end{bmatrix}, \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \right), \left(\begin{bmatrix} z_{1_\beta} \\ \vdots \\ z_{m_\beta} \end{bmatrix}, \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix} \right) \right)$$

$$s_j = \left(\left(\begin{bmatrix} z_{1_\alpha} \\ \vdots \\ z_{m_\alpha} \end{bmatrix}, \begin{bmatrix} \alpha_{b1} \\ \vdots \\ \alpha_{bm} \end{bmatrix} \right), \left(\begin{bmatrix} z_{1_\beta} \\ \vdots \\ z_{m_\beta} \end{bmatrix}, \begin{bmatrix} \beta_{b1} \\ \vdots \\ \beta_{bm} \end{bmatrix} \right) \right)$$

Now to satisfy condition (ii) of theorem 3.3.2 we need only to show that within s_j, $\alpha_{bp} = \alpha_{bq}$ and $\beta_{bp} = \beta_{bq},\ \forall p, q \in [1, m]$,. But looking at the example above, we can see that this is equivalent to showing that the $\bar{k}$ vectors of s_i, s_j are linearly dependent, which mirrors the original process we went through to find the original state vectors. So let's create another 2-tuple with the transpose of the $\bar{k}$ vectors stacked on top of

each other, which we will denote $(\bar{k}A, \bar{k}B)$.

$$\left(\begin{bmatrix} 1 & \cdots & 1 \\ \alpha_{b1} & \cdots & \alpha_{bm} \end{bmatrix}, \begin{bmatrix} 1 & \cdots & 1 \\ \beta_{b1} & \cdots & \beta_{bm} \end{bmatrix} \right)$$

And for no particularly leading reason at all, let's now assume that this is in fact a behavior in that $\forall p \in [1, m],\ \alpha_{bp} = \alpha_b,\ \beta_{bp} = \beta_b$ The new 2-tuple will look like the following.

$$\left(\begin{bmatrix} 1 & \cdots & 1 \\ \alpha_b & \cdots & \alpha_b \end{bmatrix}, \begin{bmatrix} 1 & \cdots & 1 \\ \beta_b & \cdots & \beta_b \end{bmatrix} \right)$$

If we run this through the process $g(\bar{k}A, \bar{k}B)$, the result will be

$$\left(\left(\begin{bmatrix} z_1 \\ z_1 \end{bmatrix}, \begin{bmatrix} 1 \\ \alpha_b \end{bmatrix} \right), \left(\begin{bmatrix} z_1 \\ z_1 \end{bmatrix}, \begin{bmatrix} 1 \\ \beta_b \end{bmatrix} \right) \right)$$

This shows that the scaling vectors were in fact parallel, and the scaling factors were (α_b, β_b). And so we see that condition (ii) of theorem 3.3.2 is satisfied and hence we can define a behavior $\pi = s_i$. What is even nicer is that the final scalars we just found can be used to scale π such that $s_j = (\alpha_b, \beta_b)\pi$.

And if we had previously assumed the contrary, that for some $p \in [1, m],\ \alpha_{bp} \neq \alpha_b,\ \beta_{bp} \neq \beta_b$, then the result of $g(\bar{k}A, \bar{k}B)$ would have shown different states, as they are no longer linearly dependent.

So we can make a final definition for determining whether or not two strings of MDLe_a calls represent the same behavior.

Theorem 3.3.8. *Given two strings of merged atoms in 2-tuple form $s_i = (A_i, B_i)$, $s_j = (A_j, B_j)$, $\exists \pi$ such that $\pi = s_i$ iff:*

(i) Given $g(A_i, B_i) = ((A_{i_{\bar{z}}}, A_{i_{\bar{k}}}), (B_{i_{\bar{z}}}, B_{i_{\bar{k}}}))$ and $g(A_j, B_j) = ((A_{j_{\bar{z}}}, A_{j_{\bar{k}}}), (B_{j_{\bar{z}}}, B_{j_{\bar{k}}}))$:

$A_{i_{\bar{z}}} = A_{j_{\bar{z}}}$ and $B_{i_{\bar{k}}} = B_{j_{\bar{k}}}$

(ii) Given $\bar{k}A = \begin{bmatrix} A^T_{i_{\bar{k}}} \\ A^T_{j_{\bar{k}}} \end{bmatrix}$, $\bar{k}B = \begin{bmatrix} B^T_{i_{\bar{k}}} \\ B^T_{j_{\bar{k}}} \end{bmatrix}$ *and* $g(\bar{k}A, \bar{k}B) = ((\bar{k}A_{\bar{z}}, \bar{k}A_{\bar{k}}), (\bar{k}B_{\bar{z}}, \bar{k}B_{\bar{k}}))$*:*

$$\bar{k}A_{\bar{z}}, \bar{k}B_{\bar{z}} \textit{is parallel to} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

If (i) and (ii) hold, then $\pi = s_i$, $(\alpha_b, \beta_b) = (\bar{k}A_{\bar{k}}(2,1), \bar{k}B_{\bar{k}}(2,1))$, *where* $\bar{k}X(x,y)$ *denotes element in row* x *and column* y*. Thus*

$$s_j = (\alpha_b, \beta_b)\pi$$

3.3.2 Building the Alphabet

We can now develop a theoretical algorithm to build a behavioral alphabet based on an MDLe$_a$ string. The purpose of this algorithm is to identify and store all of the valid behaviors that exist in an MDLe$_a$ call. From the theory of section 3.3.1 we see that it is easiest to work in the 2-tuple form of MDLe$_a$, and so all behaviors will be stored as

$$\pi_i = (A_i, B_i)$$

where (A_i, B_i) correspond to the defining string s_i of theorem 3.3.8.

To begin, assume we are given an MDLe$_a$ string (A, B). The first step we must take is to identify the states through the process defined in 3.3.7.

$$g(A, B) = ((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$$

This will allow us to find all common subsequences in $A_{\bar{z}}$ and $B_{\bar{z}}$ that potentially correspond to behaviors. We can store these identified subsequences in a set

$$\{s_1, \ldots s_n\}$$

We must note that although these subsequences are unique, there may be multiple strings corresponding to the subsequence. ie

$$s_i = \{(A_1, B_1), \ldots, (A_m, B_m)\}$$

So $\forall i, j \in [1, m],\ i \neq j$, we need to examine the corresponding 2-tuples to identify possible behaviors that do not already exist in Σ_b.

The first step is to get the $\bar{k}A, \bar{k}B$ matrices. These will already be defined by the first, $g(A, B)$ call, but we'll reiterate.

$$\begin{aligned} g(A_i, B_i) &= ((A_{i_{\bar{z}}}, A_{i_{\bar{k}}}), (B_{i_{\bar{z}}}, B_{i_{\bar{k}}})) \\ g(A_j, B_j) &= ((A_{j_{\bar{z}}}, A_{j_{\bar{k}}}), (B_{j_{\bar{z}}}, B_{j_{\bar{k}}})) \end{aligned}$$

And so

$$\bar{k}A = \begin{bmatrix} - & A^T_{i_{\bar{k}}} & - \\ - & A^T_{j_{\bar{k}}} & - \end{bmatrix} \quad \bar{k}B = \begin{bmatrix} - & B^T_{i_{\bar{k}}} & - \\ - & B^T_{j_{\bar{k}}} & - \end{bmatrix}$$

We then need to run the process again

$$g(\bar{k}A, \bar{k}B) = ((\bar{k}A_{\bar{z}}, \bar{k}A_{\bar{k}}), (\bar{k}B_{\bar{z}}, \bar{k}B_{\bar{k}}))$$

If the values of $\{A_{i_{\bar{z}}}, A_{j_{\bar{z}}}, B_{i_{\bar{z}}}, B_{j_{\bar{z}}}, \bar{k}A_{\bar{z}}, \bar{k}B_{\bar{z}}\}$ pass theorem 3.3.8 then we have found a valid behavior

$$\pi_n = (A_i, B_i)$$

We then need to check that it does not already exist in Σ_b, ie $\nexists(\alpha_b, \beta_b)\ st\ (\alpha_b, \beta_b)\pi_n \in \Sigma_b$. If this is true and it doesn't already exist, then we add it to the alphabet

$$\Sigma_b = \Sigma_b \oplus \pi_n$$

By repeating this process for all unique common subsequences in the MDLe_a call, we can identify all of the unique behaviors that existed in the MDLe_a string. To conclude this section, a detailed algorithm is given in figure 7.

INPUT: $MDLe_a = (A, B)\ , \Sigma_b$;
begin
  run $g(A, B)$;
  identify common subsequences $\{s_1, \ldots s_n\}$;
  while $p \leq n$ **do**
    get all $s_p = \{(A_1, B_1), \ldots, (A_m, B_m)\}$;
    while $\forall i, j \leq m,\ i \neq j$ **do**
      run $g(A_i, B_i)$;
      run $g(A_j, B_j)$;
      build $\bar{k}A, \bar{k}B$;
      run $g(\bar{k}A, \bar{k}B)$;
      set $= \{A_{i_{\bar{z}}}, A_{j_{\bar{z}}}, B_{i_{\bar{z}}}, B_{j_{\bar{z}}}, \bar{k}A_{\bar{z}}, \bar{k}B_{\bar{z}}\}$;
      if *set passes theorem 3.3.8* **then**
        $\pi_n = (A_i, B_i)$;
        if $\nexists(\alpha_b, \beta_b)$ *st* $(\alpha_b, \beta_b)\pi_n \in \Sigma_b$ **then**
          $\Sigma_b = \Sigma_b \oplus \pi_n$;
        end
      end
      $p = p + 1$;
    end
  end
end
OUTPUT: Σ_b

Figure 7: Theoretical Algorithm for Building an MDLe Behavioral String

3.3.3 Using the Alphabet

Suppose we are given an MDLe$_a$ string (A, B) and a behavioral alphabet

$$\Sigma_b = \{(A_1, B_1), \ldots, (A_n, B_n)\}$$

We would now like to identify sequences in (A, B) that correspond to the previously defined behaviors in Σ_b.

This is akin to finding the same sequences of linearly dependent row vectors defined in Σ_b if they exist in (A, B). So for each behavior $\pi_i = (A_i, B_i)$, we can start by creating a stacked matrix

$$(\tilde{A}_i, \tilde{B}_i) = \left(\begin{bmatrix} A_i \\ A \end{bmatrix}, \begin{bmatrix} B_i \\ B \end{bmatrix} \right)$$

We can then run the process $g(\tilde{A}_i, \tilde{B}_i) = ((\tilde{A}_{i_{\bar{z}}}, \tilde{A}_{i_{\bar{k}}}), (\tilde{B}_{i_{\bar{z}}}, \tilde{B}_{i_{\bar{k}}}))$. These matrices are equivalent to

$$g(\tilde{A}_i, \tilde{B}_i) = \left(\left(\begin{bmatrix} A_{i_{\bar{z}}} \\ A_{\bar{z}} \end{bmatrix}, \begin{bmatrix} A_{i_{\bar{k}}} \\ A_{\bar{k}} \end{bmatrix} \right), \left(\begin{bmatrix} B_{i_{\bar{z}}} \\ B_{\bar{k}} \end{bmatrix}, \begin{bmatrix} B_{i_{\bar{k}}} \\ B_{\bar{k}} \end{bmatrix} \right) \right)$$

So if the behavior π_i is in the MDLe$_a$ string, then both $A_{i_{\bar{z}}}$ and $B_{i_{\bar{z}}}$ will correspond to two submatrices (A_j, B_j) such that $(A_{j_{\bar{z}}}, A_{j_{\bar{z}}}) \subseteq (A_{\bar{z}}, B_{\bar{z}})$ and $A_{i_{\bar{z}}} = A_{j_{\bar{z}}}$, $B_{i_{\bar{z}}} = B_{j_{\bar{z}}}$. This condition in fact satisfies condition (i) of theorem 3.3.8, meaning it is a potential behavior.

We then need to create the matrices

$$\bar{k}A = \begin{bmatrix} - & A_{i_k}^T & - \\ - & A_{j_k}^T & - \end{bmatrix} \quad \bar{k}B = \begin{bmatrix} - & B_{i_k}^T & - \\ - & B_{j_k}^T & - \end{bmatrix}$$

and run the process

$$g(\bar{k}A, \bar{k}B) = ((\bar{k}A_{\bar{z}}, \bar{k}A_{\bar{k}}), (\bar{k}B_{\bar{z}}, \bar{k}B_{\bar{k}}))$$

If the values of $\{\bar{k}A_{\bar{z}}, \bar{k}B_{\bar{z}}\}$ pass condition (ii) of theorem 3.3.8, then we have found a valid behavior, as well as the scalars (α_b, β_b).

```
INPUT: MDLe_a = (A, B) , Σ_b;
begin
    MDLe_b = MDLe_a;
    while for each π_i ∈ Σ_b do
        build (Ã_i, B̃_i);
        run g(Ã_i, B̃_i);
        if ∃j st A_{i_z̄} = A_{j_z̄}, B_{i_z̄} = B_{j_z̄} then
            build k̄A, k̄B;
            run g(k̄A, k̄B);
            if g(k̄A, k̄B) passes theorem 3.3.8 then
                (A_j, B_j) = (α_b, β_b)π_i;
                Replace (A_j, B_j) in MDLe_b;
            end
        end
    end
end
OUTPUT: Σ_b
```

Figure 8: Theoretical Algorithm for Building an MDLe Behavioral String

Thus we can replace the original MDLe_a calls in (A, B) corresponding to the submatrices (A_j, B_j) with the call

$$(A_j, B_j) = (\alpha_b, \beta_b)\pi_i$$

By performing this action for every $\pi_i \in \Sigma_b$, we can find all of the MDLe_a calls that can be replaced by behaviors. We conclude this section with a detailed algorithm, shown in figure 8.

CHAPTER IV

IMPLEMENTATION OF THEORY FOR EXPERIMENT

In chapter 3 we successfully developed a theoretical algorithm for the construction and use of atom and behavioral alphabets. The development of these constructs was based on our initial premise that if we built these alphabets using the method of bottom-up mimicry, they would be able to create any signal related to a high-level task.

But this method of alphabet construction is still just a hypothesis, and so the theory just developed might not be worth anything if the premise proves false. I hope this isn't true, but at this point we only have a conjecture. So what we need to do now is build a MATLAB program that can implement the theory of chapter 3 in the real world so that we can test whether this idea of bottom-up mimicry is valid.

In this chapter, we will do just that. We will begin with a brief overview of the structure and use of the program, and then delve into some of the details regarding each major processing unit.

4.1 Overview of Implementation

There are two separate tasks a user can choose to perform on a given data set of control signals with this program. The first function will augment or build a set of given atom and behavioral alphabets (Σ_a, Σ_b) to replicate all signals in the data set. The second function will decompose the signals into separate MDLe_a and MDLe_b strings based on supplied alphabets. This splitting of tasks gives the user a powerful tool to experiment with various data set sizes to build the alphabets, as well as analyzing how the alphabet performs with control signals that were not used to build the alphabets. The overall structure of the program can be seen in figure 9.

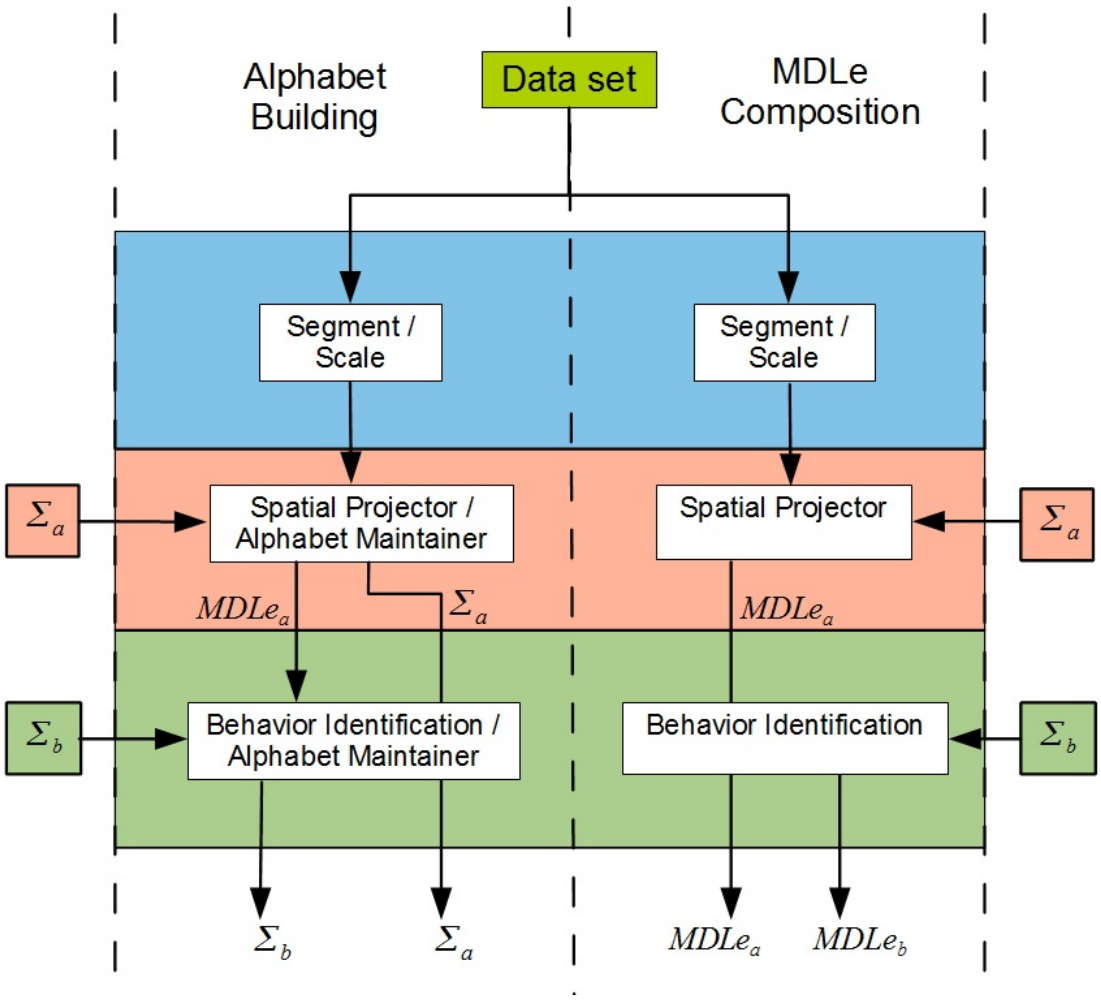


Figure 9: Overview of Program

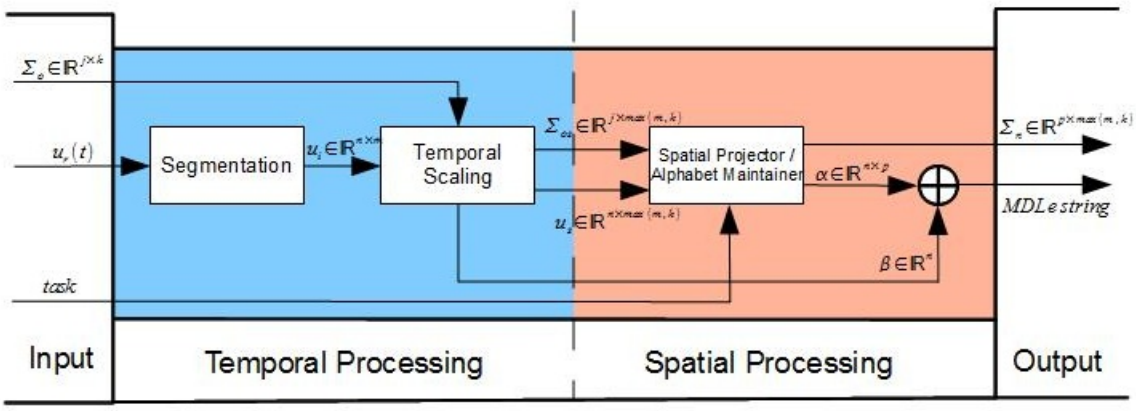


Figure 10: Overview of Atom Processor

As you can see, there is a nice similarity between these two processes. So much in fact that the program uses the exact same functions for the true processing of the theory and only adds or suppresses tasks and outputs based on the overall task being performed. The specific details of this will be conveyed in the next two sections.

4.2 Atom Processor

The purpose of the atom processor is to implement the theory of section 3.2 to build MDLe_a strings from a control signal, and augment the alphabet Σ_a when the user is building an alphabet. An overview of the process performed on each signal of the data set is given in figure 10

4.2.1 Overview

To begin, there are three inputs to the system. The first is the control signal $u_r(t),\ t \in [0, T]$ we wish to decompose. The second input is an existing MDLe_a alphabet $\Sigma_o \in \mathbb{R}^{j \times k}$, where j is the number of atoms contained in Σ_o, and k is the number of samples in each of the j atoms. In the case that $j = 0$, i.e. the alphabet is empty, the algorithm will build a new alphabet from scratch.

The final input is a task marker to let the spatial projector know whether we are

building an alphabet, or just constructing MDLe$_a$ strings.

Upon entering the algorithm, the first step is the temporal processing unit. First, $u_r(t)$ is segmented into a matrix $u_i \in \mathbb{R}^{n\times m}$, where n is the number of segments identified, and m is the number of samples in the longest segment. These signals, along with the alphabet Σ_o, are then sent to the scaler. Here, both the matrix u_i and Σ_o are temporally scaled to the length of the longest segment in $u_i \cup \Sigma_o$ and turn into the matrices $u_s \in \mathbb{R}^{n\times max(m,k)}$ and $\Sigma_{os} \in \mathbb{R}^{j\times max(m,k)}$. This step is essential because now all of the signals in u_i and Σ_{os} exist in the same time frame $t \in [0, T]$ and can thus exist in the same $L_2[0, T]$ Hilbert space, which is required by the spatial processor to use the theory developed in chapter 3.2. This step also produces our time-scaling vector $\beta \in \mathbb{R}^n$ which will allow us to rescale all of the segmented signals back to their original length.

With the temporal processing finished, u_i and Σ_{os} are sent to the spatial processor. It is here that we analyze each signal using the theory developed in chapter 3.2 to find the parallel combination of atoms from Σ_{os} that represent the projection of u_i onto Σ_{os}, and to augment Σ_{os} when instructed by the task input.

After all of the signals in u_i have been processed, the algorithm is finished and outputs two items. The first is our potentially augmented alphabet $\Sigma_n \in \mathbb{R}^{p\times max(m,k)}$, where p is the number of atoms in our MDLe$_a$ alphabet, and $max(m, k)$ is the number of samples in each atom.

The second output is a MATLAB cell containing two matrices representing the MDLe$_a$ call in 2-tuple form as defined in 3.3.1.

4.2.2 Temporal Processor

The first main unit of the algorithm is the temporal processing unit. This processor is comprised of two separate functions: the segmentation function and the scaling function. The overall purpose of this unit is to segment a given control $u_r(t)$ into a

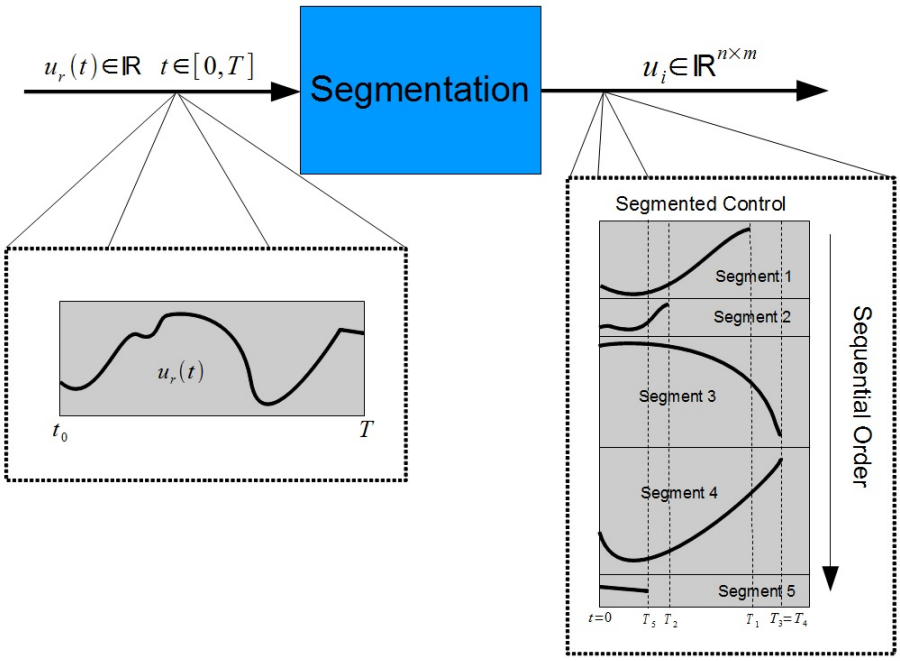


Figure 11: Temporal Segmentation of Signal

group of n individual signals, and then scale them, along with the alphabet Σ_o, to a common time frame so they can coexist in the same $L_2[0,T]$. I will now proceed to explain in more detail both of the functions contained within this unit.

4.2.2.1 Segmentation Function

The overall function of the segmentation unit is shown in figure 11. The input to this function is the original control signal $u_r(t),\ t \in [0,T]$. In reality, this is a sampled signal existing in a vector $u_r \in \mathbb{R}^{1 \times d}$ with all d samples spaced at intervals of $\triangle t$. After processing by the unit, the output is a matrix $u_i \in \mathbb{R}^{n \times m}$ consisting of n segmented signals on each row vector, and m samples in each signal.

Within the segmentation unit there are two subsequent steps. The first step is to identify important points in the signal $u_r(t)$ that signify the beginning and end of individual segments.

We do this using a discrete version of the method presented in chapter 3.1.1, with

a slight modification. As in that method, we first get the second derivative.

$$\begin{cases} u_r(k) & & , & k \in [1, d] \\ \dot{u}_r(k) & = & \dfrac{u_r(k+1) - u_r(k)}{\triangle t} & , & k \in [1, d-1] \\ \ddot{u}_r(k) & = & \dfrac{\dot{u}_r(k+1) - \dot{u}_r(k)}{\triangle t} & , & k \in [1, d-2] \end{cases} \tag{13}$$

With $\ddot{u}_r(k)$ calculated, we have provided two options for transition point identification the user can toggle between for comparison. The first approach is to identify the inflection points as discussed in chapter 3.1.1.

The second approach is to identify transition points based purely on whether the signal is moving from a period of non-zero acceleration to zero acceleration, and vise versa. The 4 conditions possible, and results are given in equation (14).

$$\begin{cases} ||\ddot{u}_r(k-1)|| > \epsilon & \&\& & ||\ddot{u}_r(k)|| > \epsilon & \rightarrow & segment\ end = FALSE \\ ||\ddot{u}_r(k-1)|| > \epsilon & \&\& & ||\ddot{u}_r(k)|| < \epsilon & \rightarrow & segment\ end = TRUE \\ ||\ddot{u}_r(k-1)|| < \epsilon & \&\& & ||\ddot{u}_r(k)|| > \epsilon & \rightarrow & segment\ end = TRUE \\ ||\ddot{u}_r(k-1)|| < \epsilon & \&\& & ||\ddot{u}_r(k)|| < \epsilon & \rightarrow & segment\ end = FALSE \end{cases} \tag{14}$$

In equation (14), ϵ is a scalar boundary defined by the user to avoid segmenting moves that only look like they are moving due to numerical bouncing around 0.

After either of these approaches are used to identify the segment transition points, the signal is passed to a physical segmenting function which cuts up the signal $u_r(k)$ into the output matrix of the segmentation unit, $u_i \in \mathbb{R}^{n \times m}$.

4.2.2.2 Scaling Function

The overall function of the scaling function is shown in figure 12. There are two inputs to the function, shown on the left in figure 12. The first is $u_i \in \mathbb{R}^{n \times m}$, the matrix consisting of the segmented signals found in section 4.2.2.1. The second input is $\Sigma_o \in \mathbb{R}^{j \times k}$, the matrix containing an existing alphabet of j atoms, with each atom containing k samples.

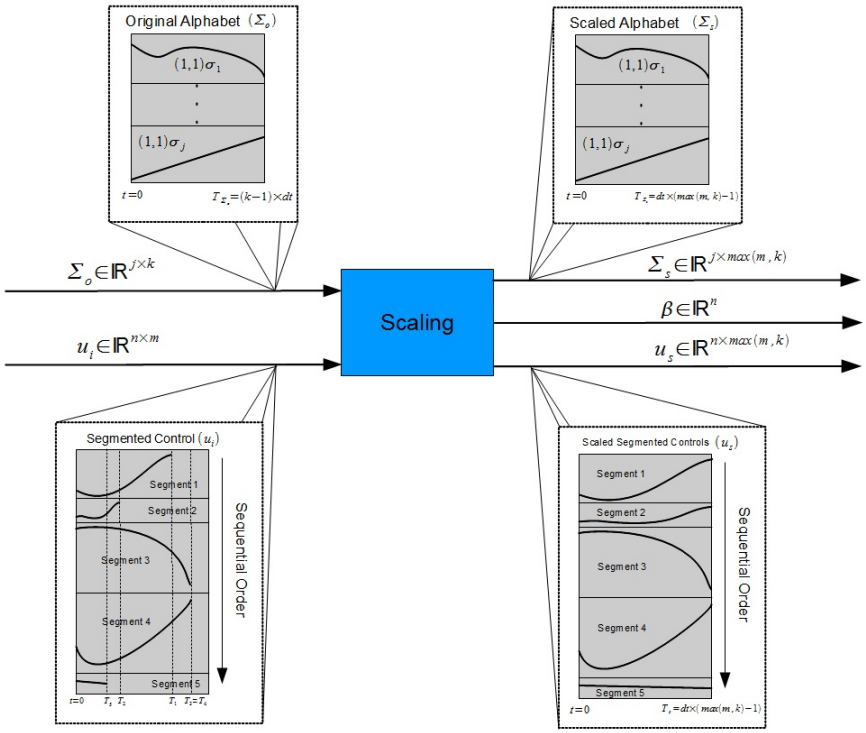


Figure 12: Temporal Scaling of Segmented Signals and Alphabet

Upon entering the function, we must first find a time period for which all of the signals in Σ_o and u_i should be temporally scaled to. This is essential to allow all of the signals to exist in the same $L_2[0,T]$ Hilbert space. In this algorithm I have chosen to scale all of the signals to the longest segment in $\Sigma_o \cup u_i$. The reasoning for this is to prevent any frequency information from being lost due to a temporal compression. In terms of the inputs, this will set the output sample size of each signal to $max(m,k)$, corresponding to a time period of $T = \triangle t \times (max(m,k) - 1)$.

With the final T established, we must now scale each signal to this time period, and then re-sample it at $\triangle t$ to maintain consistency and get the $max(m,k)$ samples required. The first step in doing this is to find the temporal scalar such that

$$\beta_s T_o = T$$

We know that $T_o = (s-1) \cdot \triangle t_s$, where s is the number of samples in the signal u_o

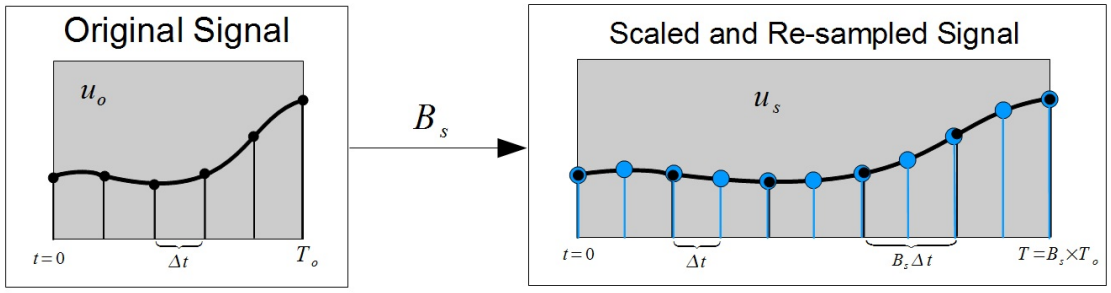


Figure 13: Temporal Scaling and Re-Sampling of a Signal

to be temporally scaled. Further, we know that $T = (max(m,k) - 1) \cdot \triangle t$. Thus,

$$\beta_s = \frac{(max(m,k) - 1) \cdot \triangle t}{(s-1) \cdot \triangle t_s}$$

And in this algorithm we will remain vigilant about keeping all of the sampling periods, meaning that $\triangle t_s = \triangle t$, and thus

$$\beta_s = \frac{(max(m,k) - 1)}{(s-1)} \tag{15}$$

With this value of β_s, we can re-scale the samples such that the interval between samples is now $\beta_s \cdot \triangle t$. We then perform a cubic spline interpolation on these samples and re-sample the interpolation at intervals of the original $\triangle t$ to get the $max(m,k)$ samples we need, while retaining the structure of the original signal. This procedure is shown in figure 13.

This procedure is performed on every signal in u_i, and every atom in Σ_o. When finished, we will have obtained the outputs of the scaling function, as shown in figure 12. The first output is the scaled alphabet $\Sigma_{os} \in \mathbb{R}^{j \times max(m,k)}$. The second output is the matrix of scaled control segments, $u_s \in \mathbb{R}^{n \times max(m,k)}$.

The final output is a vector

$$\beta \in \mathbb{R}^n \text{ st } \beta_i = \frac{1}{\beta_{s_i}}$$

This vector consists of the inverses of each β_s scalar that was used to scale each of the n segments in u_i to the time period T. We fill it with the inverses to tell the MDLe

processor how to de-scale our alphabet projections, which will be in the time period T, to the original time period T_o for each signal.

4.2.3 Spatial Processing

The purpose of the spatial processor is to find a set of merged atoms using the theory developed in section 3.2.1.3 to reproduce every segmented control found by the temporal processing unit, and to augment the alphabet if the user is building an alphabet. As shown in figure 10, there are two inputs to the system. The first is the matrix of segmented and scaled controls, $u_s \in \mathbb{R}^{n \times max(m,k)}$. The second input is the matrix of scaled alphabet atoms, $\Sigma_{os} \in \mathbb{R}^{j \times max(m,k)}$. Upon entering the spatial processor, every signal in u_s is subsequently subjected to the loop shown in figure 14.

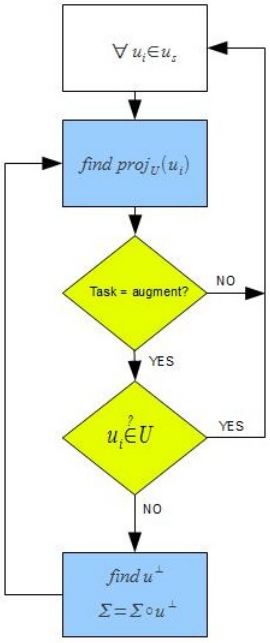


Figure 14: Main Spatial Processing Loop

4.2.3.1 Projecting and Testing

The first step shown in figure 14 is to find the projection of u_i onto Σ_{os}.

To begin we must first calculate the vector v.

$$v = \begin{bmatrix} \langle u_i, u_1 \rangle \\ \vdots \\ \langle u_j, u_n \rangle \end{bmatrix}, v \in \mathbb{R}^j \tag{16}$$

This is the vector of inner products between the signal u_i we are analyzing, and every atom in Σ.

The continuous time $L_2[0, T]$ inner product is defined as

$$\langle u, v \rangle = \int_0^T u(t)v(t)dt$$

But since we only have samples of these signals, we need to approximate it. In this algorithm, we do this using a trapezoidal numerical integration shown in equation (17)

$$\begin{cases} h(s) & = & u(s)v(s) \ , \ s \in [1, max(m, k)] \\ \langle u, v \rangle & \approx & \frac{dt}{2} \sum_{i=1}^{max(m,k)-1} (h(i+1) + h(i)) \end{cases} \tag{17}$$

After calculating v, we then calculate the α vector, containing the scalars of the $proj_U(u_i)$.

$$\alpha = G^{-1}v \quad , \alpha \in \mathbb{R}^j \tag{18}$$

As discussed earlier in section 3.2.1, G is the *Gram matrix* of inner products from Σ using the integration technique of equation (17).

If our task is to only construct the MDLe_a strings, we can exit the inner loop and move on to find the next signal segment's projection onto Σ_a. But if we are building an alphabet, then we must calculate the magnitude of the signal and run a modified version of the test described in theorem 3.2.2 to determine whether this signal $u_i \in U$.

$$|(||u_r||^2 - \alpha^T G \alpha)| < \epsilon \tag{19}$$

Here, ϵ is a scalar limit set by the user to avoid misidentifying signals due to numerical errors.

In the case that equation (19) is true, then the loop of figure 14 is terminated, the α vector is passed off to the output matrix, and the loop is repeated for the signal in u_s.

On the other hand, if the condition in (19) is not met, then we must move on to the next step of calculating the orthogonal vector and augmenting the alphabet with it.

4.2.3.2 Alphabet Augmentation

Due to the structure of Σ and u_i, calculating the orthogonal component and augmenting the existing alphabet is very easy. We first calculate $u_i^{\perp}$.

$$\begin{gathered} u_i^{\perp}(s) = u_i(s) - \alpha^T \Sigma_o(s) \ \ , \ u_i^{\perp} \in \mathbb{R}^{1 \times max(m,k)} \\ \text{where } \Sigma_o(s) \in \mathbb{R}^j \ \ , \ s \in [1, max(m,k)] \end{gathered} \tag{20}$$

Here, $\Sigma_o(s)$ is a column vector containing the desired sample from every atom in the alphabet.

After calculating $u_i^{\perp}$, we augment the alphabet by appending it after the last atom in Σ_o.

$$\Sigma_n = \begin{bmatrix} \Sigma_o \\ u_i^{\perp} \end{bmatrix} \ , \ \Sigma_n \in \mathbb{R}^{(j+1) \times max(m,k)} \tag{21}$$

We then recalculate the *Gram matrix* for the new alphabet Σ_n, and proceed back to the beginning of the loop to calculate a new projection of u_i on to the space defined by Σ_n, as shown in figure 14.

4.2.3.3 Detailed Algorithm

Shown in figure 15 is a detailed algorithm summarizing the discussion of the atom processing unit.

INPUT: $u_r \in \mathbb{R}^{1\times d}$; $\Sigma_o \in \mathbb{R}^{j\times k}$; task;
begin
  segment$(u_r) \rightarrow u_i \in \mathbb{R}^{n\times m}$;
  scale$(u_i, \Sigma_o) \rightarrow u_s \in \mathbb{R}^{n\times max(m,k)}, \Sigma_{os} \in \mathbb{R}^{j\times max(m,k)}, B \in \mathbb{R}^n$;
  compute *Gram matrix* $G \in \mathbb{R}^{j\times j}$;
  $A = [\,]$ $\Sigma_n = \Sigma_{os}$;
  while *signals still to project* **do**
    while !*done* **do**
      compute v as defined in (16);
      compute $\alpha = G^{-1}v$ as defined in (18);
      compute $||u_i||^2$;
      if $|||u_i||^2 - \alpha^T G\alpha < \epsilon \mid OR\ task = MDLe_a\ only$ **then**
        $A = \begin{bmatrix} A \\ \alpha^T \end{bmatrix}$;
        $done = TRUE$;
      end
      else
        compute $u_i^{\perp}$ as defined in (20);
        augment alphabet $\Sigma_n = \begin{bmatrix} \Sigma_n \\ u_i^{\perp} \end{bmatrix}$;
        recompute G;
        $done = FALSE$;
      end
    end
  end
  MDLe$_a$= (A,B)
end
OUTPUT: MDLe$_a$, Σ_n

Figure 15: Detailed Algorithm

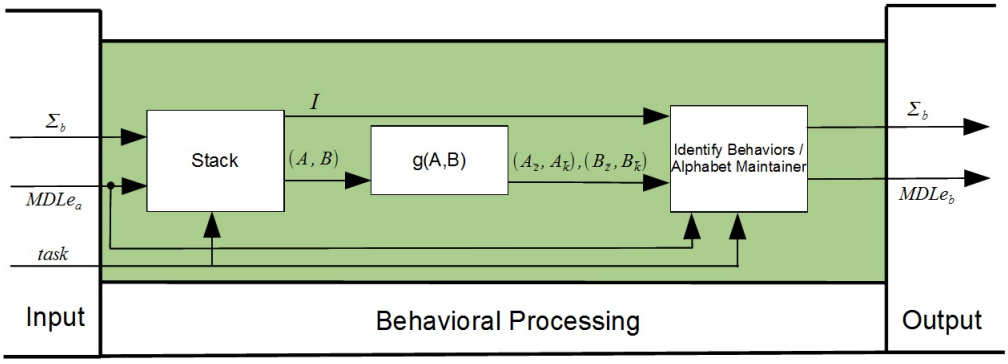


Figure 16: Overview of Behavioral Processor

4.3 Behavioral Processor

Similar to the atom processor, the behavioral processor uses very similar internal processes whether we are augmenting the alphabet, or building MDLe$_b$ strings. The general layout of the processor is shown in figure 16.

4.3.1 Overview

There are three inputs to the behavioral processor. The first is a data set of n MDLe$_a$ 2-tuples (A_i, B_i) created by the atom processor representing n distinct control signals. The second input is the behavioral alphabet $\Sigma_b = \{\pi_1, \ldots \pi_m\}$, consisting of m behaviors such that $\pi_i = (A_i, B_i)$. The final input is the task marker to let the processing units know whether the user wants to build an alphabet, or turn the MDLe$_a$ strings into MDLe$_b$ strings

Upon entering the algorithm, both Σ_b and the MDLe$_a$ strings are sent to the stacking unit in preparation for the $g(A, B)$ state identifying process. If the user is building an alphabet, then the unit stacks all of the n distinct 2-tuples from the data set into a single 2-tuple $(\tilde{A}, \tilde{B})$, and forms a vector of the end indexes for each data set. If the user is trying to create MDLe$_b$ strings, we initially create two separate 2-tuples. The first contains all of the behaviors stacked on top of each other, and the second contains all of the MDLe$_a$ 2-tuples stacked on top of each other. We then

create the output 2-tuple by stacking the behaviors on top of the MDLe$_a$ calls.

These stacked 2-tuples are then sent to the state identifying processor. This processor performs the process $g(A, B)$ defined in 3.3.7 on the 2-tuple it is passed, regardless of the overall task being performed. The output of the process is a 2-tuple of 2-tuples $((A_{i_{\tilde{z}}}, A_{i_{\tilde{k}}}), (B_{i_{\tilde{z}}}, B_{i_{\tilde{k}}}))$ consisting of the states and scalars.

Finally, these state and scalar tuples are sent to the behavior identifier. If the user is building an alphabet it will identify all behaviors from the single stacked 2-tuple $(\tilde{A}, \tilde{B})$ that do not straddle the boundaries between the distinct data sets identified by the index vector. It will then create a new alphabet Σ_b consisting of the identified behaviors. If the user is creating MDLe$_b$ strings, then the identifier will look at the 2-tuples corresponding to each behavior identified in I to see if the behavior exists within each MDLe$_a$ call. If it does, it replaces the atom calls with the corresponding behavioral call.

There are two potential outputs of the behavioral processor. The first is the alphabet Σ_b, which will always be an output. The second output is the MDLe$_b$ strings, which are created only when the user is attempting to create those strings. The MDLe$_b$ string is similar in nature to the MDLe$_a$ string, but has two added matrices.

$$MDLe_b = (M, A, BEH, B)$$

$$M \in \mathbb{Z}^{n\times 1},\ A \in \mathbb{R}^{n\times m},\ BEH \in \mathbb{R}^{n\times p},\ B \in \mathbb{R}^{n\times 1}$$

Here, there are n sequential calls in the string. M is a vector in which each row is equal to $[0, 1]$, and marks whether the particular call is an atom level call, or a behavioral call. A is the matrix corresponding to the atom level calls and contains the spatial scalars for each sequential call as a row vector. BEH is the matrix corresponding to the behavior level calls and contains the spatial scalars for each sequential call as a row vector. Finally, B is the temporal scaling vector containing the β scalars for both the atom and behavioral calls.

We will now proceed to discuss the specifics of each sub-processor in detail.

4.3.2 Stacking

The stacking unit prepares the algorithm's inputs for state processing. For inputs, it receives an alphabet Σ_b, as well as a data set of MDLe$_a$ calls. Then, depending on the task the user is performing, the process creates a 2-tuple of stacked matrices for use by the state identifier.

4.3.2.1 Alphabet Construction

If the user is creating an alphabet, the purpose of the stacking is to build a compiled 2-tuple of all the MDLe$_a$ calls so that when we identify states, the relations of linearly dependent row vectors transcend the boundaries between the distinct data sets. This will allow us to find behaviors common to various high-level users.

We do this by stacking all of the n matrices corresponding to the n MDLe$_a$ calls into two matrices $(\tilde{A}, \tilde{B})$.

$$(\tilde{A}, \tilde{B}) = \left(\begin{bmatrix} A_1 \\ \vdots \\ A_n \end{bmatrix}, \begin{bmatrix} B_1 \\ \vdots \\ B_n \end{bmatrix} \right)$$

where (A_i, B_i) is an MDLe$_a$ call

We also create an index vector I holding the end indexes of the original MDLe$_a$ calls within $(\tilde{A}, \tilde{B})$ where the index entry

$$I_i = \sum_{j=1}^{i} (\#\text{ rows in } A_j)$$

4.3.2.2 MDLe string Construction

When constructing the MDLe$_b$ strings, our goal is to determine whether the behaviors in Σ_b exist in the MDLe$_a$ strings. So we must first create a 2-tuple of the stacked

behaviors

$$(A_{\Sigma_b}, B_{\Sigma_b}) = \left(\begin{bmatrix} A_{\pi_1} \\ \vdots \\ A_{\pi_m} \end{bmatrix}, \begin{bmatrix} B_{\pi_1} \\ \vdots \\ B_{\pi_m} \end{bmatrix} \right)$$

where (A_{π_i}, B_{π_i}) is a behavior $\in \Sigma_b$

We then create the stacked matrix of all the MDLe$_a$ calls as in the previous section.

$$(A_{MDLe_a}, B_{MDLe_a}) = \left(\begin{bmatrix} A_1 \\ \vdots \\ A_n \end{bmatrix}, \begin{bmatrix} B_1 \\ \vdots \\ B_n \end{bmatrix} \right)$$

where (A_i, B_i) is an MDLe$_a$ call

Finally, we create the output 2-tuple by stacking these two sets on top of each other.

$$(\tilde{A}, \tilde{B}) = \left(\begin{bmatrix} A_{\Sigma_b} \\ A_{MDLe_a} \end{bmatrix}, \begin{bmatrix} B_{\Sigma_b} \\ B_{MDLe_a} \end{bmatrix} \right)$$

We conclude by creating the index matrix I pointing to the ends of each behavior in A_{Σ_b}, and each MDLe$_a$ call in A_{MDLe_a}.

4.3.3 g(A,B)

This processor implements the process g(A,B) defined in 3.3.7. Its purpose is to transform the input 2-tuple into 2 2-tuples that map the linear dependencies between row vectors within the A and B matrices. This is accomplished by first assigning each row a unique state as defined in 3.3.3, and then finding the scalars relating the spatial relationship between rows sharing the same state.

To do this we need to define what it means to be linearly dependent. When two rows are linearly dependent, it means $\exists k$ such that $\alpha_i = k\alpha_j$. For our case, we will make the stipulation that $k \neq 0$.

So then if we are analyzing two non-zero vectors, how do we determine linear dependence? Well, we know that linearly dependent vectors are parallel, which means

the angle θ between them is 0. We find the angle as follows.

$$\theta = cos^{-1}\left(\frac{|\alpha_i \cdot \alpha_j|}{||\alpha_i||||\alpha_j||}\right)$$

And since two vectors only share the same state identifier if they are linearly dependent, then they are the same state if the angle between them is 0.

Further, if the two vectors are in fact the same state, then we need to find the scaling relationship between them. We can do this through projection. Suppose we want to project α_j onto α_i. We know that

$$proj_{\alpha_i}(\alpha_j) = k\alpha_i$$

where

$$k = \frac{\alpha_j \cdot \alpha_i}{\alpha_i \cdot \alpha_i}$$

And so the scalar k represents the spatial relationship between our two linearly dependent vectors.

We now have 2 tools that will allow us to find linearly dependent vectors, and their scaling relationship. So let's use these to define the process g(A,B).

4.3.3.1 The Process

We know from the definition of g(A,B) that we are running the process f(A) as defined in 3.3.6 on both the A and B matrices. Given in figure 17 is the algorithm for f(A) in detail.

We begin by setting up the output templates $A_{\bar{z}} = [0] \in \mathbb{Z}^{n\times 1},\ A_{\bar{k}} = [0] \in \mathbb{R}^{n\times 1}$ where $A_{\bar{z}}$ is the state vector and $A_{\bar{k}}$ holds the scalars. We then proceed to assign every 0 vector in A a state of 1. We then set the state marker $z = 2$, as that is the lowest unused state that has not been assigned to any vectors.

We then enter the main loop. For each row $\alpha_i \in A$, we first check to see that it hasn't previously been assigned a state. If it hasn't, then we assign it the state marker value holding the lowest unused unique state identifier. We then look at each

INPUT: $A \in \mathbb{R}^{n \times m}$;
begin
  $A_{\bar{z}} = [0] \in \mathbb{Z}^{n \times 1}$; $A_{\bar{k}} = [0] \in \mathbb{R}^{n \times 1}$;
  $i = 1$;
  while $i \leq n$ **do**
    if $\alpha_i == 0$ **then**
      $A_{\bar{z}_i} = 1$;
    end
    i = i+1;
  end
  $z = 2$;
  $i = 1$;
  while $i \leq n$ **do**
    if $A_{\bar{z}_i} = 0$ **then**
      $j = i + 1$;
      $A_{\bar{z}_i} = z$; $A_{\bar{k}_i} = 1$;
      while $j \leq n$ **do**
        $\theta = cos^{-1}\left(\frac{|\alpha_i \cdot \alpha_j|}{||\alpha_i||||\alpha_j||}\right)$;
        if $\theta < \epsilon$ **then**
          $A_{\bar{z}_j} = z$;
          $A_{\bar{k}_j} = \frac{\alpha_j \cdot \alpha_i}{\alpha_i \cdot \alpha_i}$;
        end
        $j = j + 1$;
      end
      $z = z + 1$;
    end
    $i = i + 1$;
  end
end
OUTPUT: $(A_{\bar{z}}, A_{\bar{k}})$

Figure 17: f(A) for a single matrix

vector α_j positioned lower in A than α_i. If α_j has not been assigned a state, then it is possibly parallel to α_i. So we then calculate the angle θ between α_i and α_j. If it is less than a certain threshold ϵ, we have determined they are linearly dependent. We then assign α_j the same state z as α_i, and calculate the scaling relationship between the two. After we have looked at each vector lower in A than α_i, we increase the state marker by one since the current state is no longer unique, and then move onto the next vector α_i.

By running this algorithm on each matrix A and B, we can successfully create the state and scaling vectors $((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$ required to identify behaviors.

4.3.4 Behavioral Identification

The purpose of this process is to use the state and scaling information to identify behaviors. If the user is building an alphabet, then it will identify all of the valid behaviors and create a new alphabet Σ_b containing them. If the user is creating MDLe_b strings, the process will look to see if the behaviors from Σ_b exist in any of the MDLe_a strings.

There are 4 inputs. The first is the 2-tuple $((A_{i_{\bar{z}}}, A_{i_{\bar{k}}}), (B_{i_{\bar{z}}}, B_{i_{\bar{k}}}))$, the output of $g(A, B)$. The second is the vector I defined in 4.3.2 containing the boundaries of distinct segments in the (A, B) matrices. The third input is the original MDLe_a calls. The final input is the task marker telling the processor which task the algorithm is being used for.

Regardless of the task, the most fundamental action performed in this process is the identification of common subsequences, as defined in 3.3.4. For this task I have decided to use an existing script 'substr.m' written by Mike Sheppard. This script has two functions that I have used.

(i) $x, n = sub(A)$: Finds the longest common subsequences in A and returns the length and indexes to the beginning of the subsequences, contained in n

INPUT: $MDLe_a, ((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$, I;
$\Sigma_b = \{\}$;
begin
  $n_A = sub(A_{\bar{z}})$;
  $x =$length of common subsequence found;
  while $x \geq 1$ **do**
    $n_x = sub(A_{\bar{z}}, x) \rightarrow \{s_1, \ldots, s_n\}$;
    while $i \leq n$ **do**
      $s_i \rightarrow \{((A_{1_{\bar{z}}}, A_{1_{\bar{k}}}), (B_{1_{\bar{z}}}, B_{1_{\bar{k}}})), \ldots, ((A_{m_{\bar{z}}}, A_{m_{\bar{k}}}), (B_{m_{\bar{z}}}, B_{m_{\bar{k}}}))\}$;
      for $\forall((A_{p_{\bar{z}}}, A_{p_{\bar{k}}}), (B_{p_{\bar{z}}}, B_{p_{\bar{k}}}))$ *that does not straddle boundary in* I **do**
        $\bar{k}A = \begin{bmatrix} A^T_{1_{\bar{k}}} \\ \vdots \\ A^T_{q_{\bar{k}}} \end{bmatrix}$ $\bar{k}B = \begin{bmatrix} B^T_{1_{\bar{k}}} \\ \vdots \\ B^T_{q_{\bar{k}}} \end{bmatrix}$;
        $((\bar{k}A_{\bar{z}}, \bar{k}A_{\bar{k}}), (\bar{k}B_{\bar{z}}, \bar{k}B_{\bar{k}})) = g(\bar{k}A, \bar{k}B)$;
        for *each unique state in* $\bar{k}A_{\bar{z}}$ *st all* $\bar{k}B_{\bar{z}}$ *are equal* **do**
          if *# of occurrences* > 1 **then**
            $\pi = (A_i, B_i)$;
            $\Sigma_b = \Sigma_b \oplus \pi$;
          end
        end
      end
      $i = i + 1$;
    end
    $x = x - 1$;
  end
end
OUTPUT: Σ_b

Figure 18: Process to Construct Behavioral Alphabet

(ii) $n = sub(A, x)$: Finds all common subsequences of length x.

I will now proceed to briefly describe the processes for the separate alphabet and MDLe$_b$ tasks, attempting to relieve the reader of all the fine details of the algorithm, as they are quite tedious.

4.3.4.1 Alphabet Construction

A semi-detailed version of the algorithm for constructing a behavioral alphabet is given in figure 18. For this process, we use the output of the $g(\tilde{A}, \tilde{B})$ process to

identify common behaviors within the original stacked matrices $(\tilde{A}, \tilde{B})$ we created in section 4.3.2.1. When common behaviors are identified, they will be added to the alphabet Σ_b, that is initially empty.

The first step is to run the function $n_A = sub(A_{\bar{z}})$ to identify the longest common subsequence of states in $A_{\bar{z}}$. As the length of this subsequence is the longest we will find, we use its length as the basis and iteratively search for subsequences of length $[x, x-1, \ldots, 1]$.

For each iteration, we then find all common subsequences for the specific length specified by the iteration index. This results in a set of strings $\{s_1, \ldots, s_n\}$, in which each string s_i has a set of 2-tuples $\{((A_{1_{\bar{z}}}, A_{1_{\bar{k}}}), (B_{1_{\bar{z}}}, B_{1_{\bar{k}}})), \ldots, ((A_{m_{\bar{z}}}, A_{m_{\bar{k}}}), (B_{m_{\bar{z}}}, B_{m_{\bar{k}}}))\}$ associated with the common subsequences' position and length within $A_{\bar{z}}$.

For each common subsequence s_i identified, we must weed out the instances of s_i that straddle the data set boundaries identified in I. After this, we can create the $(\bar{k}A, \bar{k}B)$ matrices to identify which vectors of scalars are actually linearly dependent, satisfying condition (ii) of theorem 3.3.8. We then run the process $g(\bar{k}A, \bar{k}B)$ which will assign states to the scaling vectors. If its output contains a set of states that occur more than one time, then we have identified a common subsequence that satisfies all of the conditions of theorem 3.3.8, and can thus be considered a behavior. We then define the behavior $\pi = (A_i, B_i)$, where A_i, B_i are the submatrices of the original MDLe$_a$ strings associated with the first instance of the behavioral subsequence identified. We then add this to our alphabet Σ_b.

4.3.4.2 MDLe$_b$ string construction

A semi-detailed version of the MDLe$_b$ string construction process is given in figure 19. Its purpose is to construct an MDLe$_b$ string for each of the n original MDLe$_a$ strings the algorithm was passed. It has 3 inputs. The first is the original set of MDLe$_a$ strings. The second is the output of the $g(\tilde{A}, \tilde{B})$ process. The final input is the index

INPUT: $MDLe_a, ((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$, I;
$((A_{i_{\bar{z}}}, A_{i_{\bar{k}}}), (B_{i_{\bar{z}}}, B_{i_{\bar{k}}})) \subseteq ((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$, $\forall (A_i, B_i) \in MDLe_a$, $i \in [1, n]$;
$((A_{\pi_{j_{\bar{z}}}}, A_{\pi_{j_{\bar{k}}}}), (B_{\pi_{j_{\bar{z}}}}, B_{\pi_{j_{\bar{k}}}})) \subseteq ((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$, $\forall \pi \in \Sigma_b$, $j \in [1, m]$;
begin
  $i = 1$;
  while $i \leq n$ **do**
    $M_i = 0^{p \times 1}$, $A_i \in \mathbb{R}^{p \times q}$, $BEH_i = 0^{p \times m}$, $B_i \in \mathbb{R}^{p \times 1}$;
    $j = 1$;
    while $j \leq m$ **do**
      if $(A_{\pi_{j_{\bar{z}}}}, B_{\pi_{j_{\bar{z}}}}) \subseteq (A_{i_{\bar{z}}}, B_{i_{\bar{z}}})$ **then**
        $\bar{k}A = \begin{bmatrix} A^T_{\pi_{j_{\bar{z}}}} \\ A^T_{i_{\bar{z}}} \end{bmatrix}$ $\bar{k}B = \begin{bmatrix} B^T_{\pi_{j_{\bar{z}}}} \\ B^T_{i_{\bar{z}}} \end{bmatrix}$;
        $((\bar{k}A_{\bar{z}}, \bar{k}A_{\bar{k}}), (\bar{k}B_{\bar{z}}, \bar{k}B_{\bar{k}})) = g(\bar{k}A, \bar{k}B)$;
        for *each of r subsequences that satisfy theorem 3.3.8 with* π_j **do**
          get (α_b, β_b) for this subsequence;
          assume matched behavior has indexes (s, e) in MDLe_{a_i};
          delete rows $(s+1, e)$ in M_i, A_i, BEH_i, B_i;
          $(A_{i_{se_{\bar{z}}}}, B_{i_{se_{\bar{z}}}}) = (NaN, NaN)$;
          $M_{i_{s1)}} = 1$;
          $A_{i_{s:}} = NaN$;
          $BEH_{i_{sj}} = \alpha_b$;
          $B_{i_{s1}} = \beta_b$;
        end
      end
    end
    $MDLe_{b_i} = (M_i, A_i, BEH_i, B_i)$;
    $i = i + 1$;
  end
end
OUTPUT: Σ_b

Figure 19: Process to Construct an MDLe_b String

vector I.

The first step in the process is to break apart the input matrices $((A_{\bar{z}}, A_{\bar{k}}), (B_{\bar{z}}, B_{\bar{k}}))$ into the respective matrices corresponding to the n MDLe$_a$ calls and m behaviors $\pi \in \Sigma_b$. The result is the set of n 2-tuples $((A_{i_{\bar{z}}}, A_{i_{\bar{k}}}), (B_{i_{\bar{z}}}, B_{i_{\bar{k}}}))$, with each 2-tuple i corresponding to an MDLe$_a$ call, and a set of m 2-tuples $((A_{\pi_{j_{\bar{z}}}}, A_{\pi_{j_{\bar{k}}}}), (B_{\pi_{j_{\bar{z}}}}, B_{\pi_{j_{\bar{k}}}}))$, with each 2-tuple corresponding to a behavior $\pi_j \in \Sigma_b$.

With these broken down matrices, we now want to look at each MDLe$_a$ call separately to see if any of the m behaviors from Σ_b are contained within the call. And since all of the states and scalars were created during the same $g(\tilde{A}, \tilde{B})$ process, these broken down matrices will still contain the linear dependence relationships between all of the vectors in both the MDLe$_a$ call and Σ_b 2-tuples.

So for each MDLe$_a$ call, we create the template for the MDLe$_b$ call where

$$MDLe_b = (M, A, BEH, B)$$

where: $M_i = 0^{p\times 1}$ is a marking vector signaling whether the values contained on the rows of A, BEH, B are behavioral calls or atom calls, initialized to all atom calls; $A_i \in \mathbb{R}^{p\times q}$ is the original atom scaling matrix A; $BEH_i = 0^{p\times m}$ will contain the spatial scaling values for use against the behavioral alphabet Σ_b; and $B_i \in \mathbb{R}^{p\times 1}$ holds the temporal scalars.

We then analyze whether each behavior π_j is in this MDLe$_a$ call. If the matrices $(A_{\pi_{j_{\bar{z}}}}, B_{\pi_{j_{\bar{z}}}})$ appear in the MDLe$_a$ matrices $(A_{i_{\bar{z}}}, B_{i_{\bar{z}}})$, then condition (i) of theorem 3.3.8 is satisfied, and thus the behavior may be present in the call.

So we create the matrices

$$\bar{k}A = \begin{bmatrix} A^T_{\pi_{j_{\bar{z}}}} \\ A^T_{i_{\bar{z}}} \end{bmatrix} \bar{k}B = \begin{bmatrix} B^T_{\pi_{j_{\bar{z}}}} \\ B^T_{i_{\bar{z}}} \end{bmatrix}$$

and run the process $g(\bar{k}A, \bar{k}B) = ((\bar{k}A_{\bar{z}}, \bar{k}A_{\bar{k}}), (\bar{k}B_{\bar{z}}, \bar{k}B_{\bar{k}}))$ to find the linear dependence relationship between the scaling vectors. If there is a 2-tuple in $(\bar{k}A_{\bar{z}}, \bar{k}B_{\bar{z}})$

such that the state assigned to the MDLe$_a$ call is equal to the state assigned to the behavior, then condition (ii) of theorem 3.3.8 is satisfied, and we have identified an instance of the behavior π_j within the call MDLe$_{a_i}$.

We then get the (α_b, β_b) scalars relating the MDLe$_a$ call to the behavior, and insert them into the BEH and B matrices at their respective locations. This is followed by changing the value in M to 1 to indicate to the MDLe parser that we are implementing a behavior, and not an atom. Finally, we delete all of the rows in the MDLe$_b$ matrices that were previously occupied by MDLe$_a$ calls, which we just obsoleted with a behavior call. The reason for this is to prevent the processor from re-assigning those rows to a different behavior that may overlap with this one.

After this process is performed for each behavior $\pi_j \in \Sigma_b$, the result will be the final MDLe$_b$ string for this piece of data. We then repeat the process for each full signal in the original data set.

CHAPTER V

EXPERIMENT

Now that we have a working MATLAB algorithm capable of building and using MDLe alphabets, we can return to the fundamental question behind this thesis: Can the method of bottom-up mimicry be used to build an alphabet of signals capable of creating any control required by a high-level user?

To test this idea, we will first need a robotic apparatus that can already perform the high-level tasks a user wants to perform. In this experiment, we will use a Khepera robot that can be driven around with a joystick.

The second thing we need is two separate groups of users performing different tasks. In this experiment, we will find two distinct groups of volunteers who are tasked with performing different driving tasks. The first group will navigate a closed path in any manner they please, while the second group will navigate a well-defined path with an exact start and end point.

With these data-sets, we can then test the method of bottom-up mimicry. First, we will use one group's data-set as a control from which to build an alphabet. We then use the other group's data to build MDLe strings based on this alphabet. We can then compare how closely the reconstructed MDLe signals approximate the original signals.

If the approximation is good, it would suggest that we have in fact developed a controls alphabet that is so rich in its ability to recreate high-level tasks, that it could be stored on a robotic apparatus, and incorporated into an MDLe system.

We will now take a brief look at the hardware used in this experiment, and then lay out a methodology to test our hypothesis, as well as some secondary items of

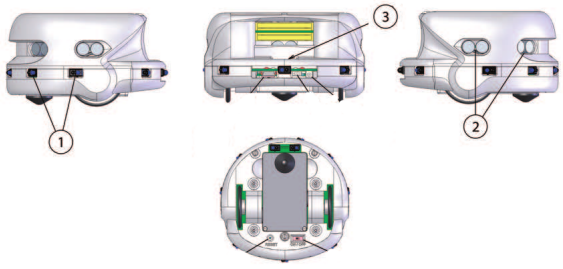


Figure 20: Illustration of Khepera Robot: 1)IR sensors 2)Ultrasonic sensors 3)Expansion Slot for Compact Flash Wireless Card

interest.

5.1 Hardware

All of the hardware used in this experiment was part of the Georgia Robotics and Intelligent Systems (GRITS) Lab, headed by my adviser Dr. Egerstedt. There were three main pieces of equipment used to run the experiment: a robot, a motion capture system, and a joystick controller.

5.1.1 Khepera Robots

The robot used in this experiment was the Khepera III robot. It is a small, modular robotic platform driven by a two-wheel differential drive. The wheel drive motors are driven via onboard, closed-loop motor controllers from two reference inputs: forward velocity and rotational velocity, giving it unicycle dynamics.

$$\begin{cases} \dot{x} = v \cdot cos(\theta) \\ \dot{y} = v \cdot sin(\theta) \\ \dot{\theta} = w \end{cases}$$

These inputs v, w will be used as the control signals we analyze with this algorithm.

As for connectivity, the KoreBot II module was used to provide a Linux OS, and

(a) Close-Up (b) To Scale

Figure 21: Khepera Robot with Vicon Motion Capture System in GRITS Lab

an 802.11g compact flash wireless card was used to give access to the LAN. This setup allowed us to SSH into the OS and run a driver that listened over the LAN for the control signals v, w.

5.1.2 Vicon Motion Capture System

The GRITS lab contains a Vicon motion capture system capable of tracking the three dimensional pose of multiple bodies simultaneously. The system works by emitting IR light from a ring around the lens of an IR camera, which is reflected back to the source by retroreflectors on a small ball attached to the object we are tracking. With our system of eight cameras working together, we are able to localize the robot to within $\pm 5mm$, with a refresh rate of 5 Hz.

For this experiment, the positional data was required by the joystick controller to calculate the required control signals v, w to run the Khepera.

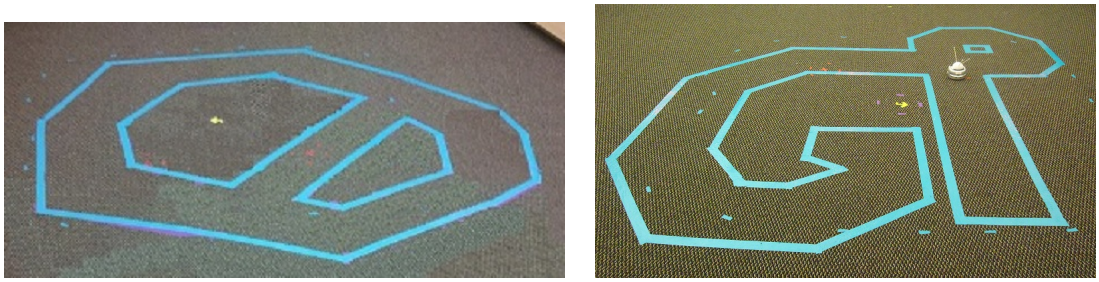
5.1.3 Joystick Controller

To allow the user to control the robot in real-time, we used an existing implementation developed by Rahul Chipalkatty. This implementation is a Lyapunov controller that combines the user's joystick data with the Vicon positional data to send a control signal that mimics the user's desired path, while ensuring system stability.

5.2 Methodology

As mentioned previously, the experimental set-up is to have 2 separate groups of users drive a Khepera robot around two distinct courses by way of joystick. I will now layout in detail the 3 major steps involved in the experiment, and describe the specific metrics we will use to analyze performance.

5.2.1 Physical Experiment



(a) Course 1 (b) Course 2

Figure 22: Course Layouts for Experiment

The purpose of the physical experiment is to get two separate groups of users to generate two data sets of control signals. Each group of users is given a different course layout, with different tasks. The hope is to get enough users for each group to ensure a strong alphabet can be built from each.

The first course layout is shown in figure 22(a). Each user will be given a starting point and told to drive through the course in any manner and path they want, as long as they end up in the same spot they started. This will be repeated 3-4 times,

with the stipulation that they take a different route on each trip.

The second course layout is shown in figure 22(b). Each user will start at either end of the path and drive to the other side. Then for each subsequent trip, they will drive back to the other side. This will be repeated 3-4 times per user.

5.2.2 Alphabet Construction

Once the data sets have been gathered we will build separate alphabets for each so we can compare the resulting MDLe strings produced using both alphabets. And as we mentioned in chapter 4.2.2.1, the MATLAB algorithm we constructed offers 2 methods of temporal segmentation that can be used to build an alphabet. We currently have no clue which is the better method, so we will want to experiment with alphabets constructed using both methods to determine the superior method of segmentation. So for each group's data set we will build 2 alphabets: one using the pure *inflection* point method, and the other segmenting the signals based on periods of *acceleration and no acceleration*.

Thus, the result will be 4 distinct alphabets that we need to build. Our hopes for these alphabets is that each is rich enough to recreate the user signals from their group, as well as the opposing group.

But if we expect the alphabets from each group to be rich enough to recreate all the user signals, then they should be able to create each other, and hence are essentially the same alphabets. So we will attempt to measure the amount of similarity between them.

5.2.2.1 Measuring Similarity Between Atom Alphabets

We know from chapter 3 that an alphabet is a subspace of a Hilbert space. Lets denote our two data sets as $G_1, G_2 \subseteq H$. If they truly are the same alphabet, then

$$G_1 \cap G_2 = G_1 = G_2$$

But this may not be the case, so we can make the following definitions

$$G_1 \cap G_2 = I$$
$$I \subseteq G_1, G_2$$

And so we are interested in the relation of I to both G_1 and G_2. A good metric for this is to find the percent of the space G_1 that is contained in G_2, and vice versa. Since G_1 is comprised of basis vectors, we can find this percentage by projecting each basis vector in G_1 onto G_2 to see if it is already contained in the space occupied by G_2, which can be determined using our previously defined theorem 3.2.2. This gives us our first metric for measuring similarity.

Definition 5.2.1. The percentage of G_x contained in G_y is called the **Overlap** and is defined as

$$O_{x \subseteq y} = \frac{dim(I)}{dim(G_x)}$$
$$st\ I = G_x \cap G_y$$

Finally, we will define a similarity percentage.

Definition 5.2.2. The percentage of space defined by the intersection of the two alphabets, to the entire space spanned by the union of the two alphabets is the **Similarity** and is defined as

$$S = \frac{dim(G_1 \cap G_2)}{dim(G_1 \cup G_2)}$$

5.2.2.2 Measuring Similarity Between Behavioral Alphabets

Remember that behavioral alphabets break the temporal segmentation barriers. So in this experiment we would like to measure the similarity between all of the behavioral alphabets that we built, because it is very possible that they contain some shared signals that are connected directly to high-level behaviors.

But when we defined our behaviors, they were not defined as spaces, but rather strings of calls. Yet each of these strings do translate to a temporal signal. So to analyze these behaviors, we will reconstruct all of these behavioral signals, scale them to the same time frame, and then build an atom alphabet based on them. This space will span all of the signals contained in the behavioral alphabets, but be defined by a minimal set of basis vectors.

With these spaces formed, we will then use the metrics of overlap and similarity defined previously to evaluate how similar our behavioral alphabets are.

But another observation on these behavioral alphabets is that they may contain a large number of behaviors that are never used. This is due to an inefficiency in my algorithm in which the number of behaviors defined for each longest length is the combination of that length. For example, if the behavior 123 is defined, the behaviors

$$123 \rightarrow \{123, 12, 23, 1, 2, 3\}$$

will also be defined, since each subsequence is also repeated multiple times in the MDLe_a calls. So we will create a modified behavioral alphabet consisting only of the behaviors that were actually called upon by any of the constructed MDLe_a strings, and compare their overlap and similarity.

5.2.3 MDLe String Construction

With the MDLe alphabets formed, we will then build the corresponding MDLe_a and MDLe_b strings for each group based on the alphabets constructed from its group's data, as well as the alphabets from the other group's data. We can then measure the performance of the recompiled strings against the original signals using a few specific metrics.

The first metric we are interested in is the average temporal length of each individual MDLe call. Our expectation is that the average length of the MDLe calls utilizing behaviors will be longer, as the behaviors are designed to group numerous

sequential atom calls into a single call.

The second metric will be the error between the recompiled MDLe strings and the original signals. Specifically we will look at the root mean square error (RMSE), and the ratio of this error to the original magnitude of the signal. This will give us a good handle on how closely the MDLe calls are approximating the original signals.

The final metric will be the bandwidth needed to transmit the MDLe strings versus the original bandwidth needed to transmit the full signals.

These metrics will then be used to make various comparisons between the alphabets built from different groups, and different temporal segmentation methods.

CHAPTER VI

RESULTS

In this chapter I will present results from the experiment described in chapter 5.

6.1 Physical Experiment Results

The collected Khepera motion paths are shown in figure 23.

The users generally stayed within the paths defined by the tape shown in figure 22. The users in course 1 took a very free form approach as the task was specifically general. This included a few users that drove backwards through the whole course, which kind of proves the point of this thesis: a designer can't predict everything a user may do. Contrary to course 1, the users in course 2 were much more structured, as the task was well defined.

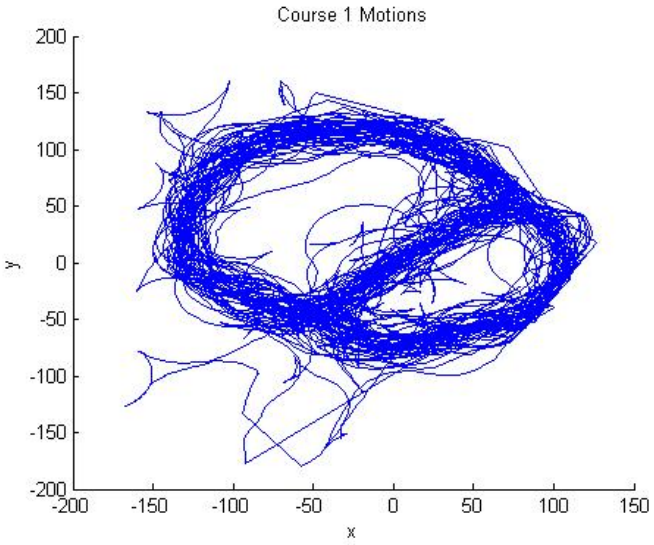
6.2 Alphabet Construction

In this section, I present data on how the atom and behavioral alphabets were constructed for each of the data sets. The basic statistics on the number of atoms and behaviors added to the alphabets is shown in table 1.

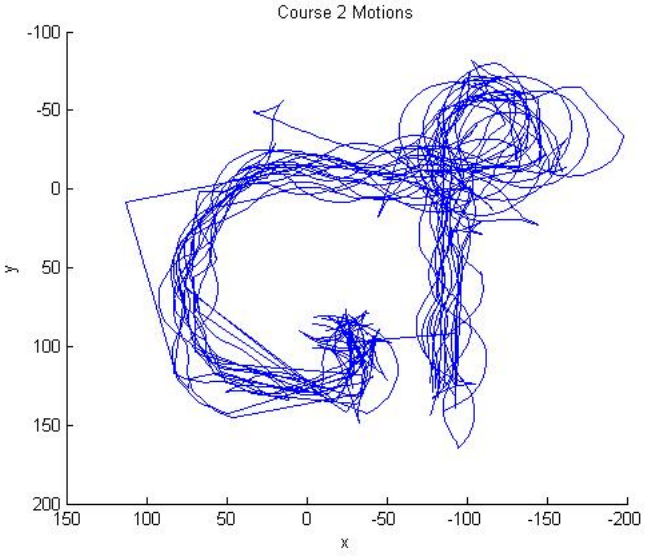
Table 1: Alphabet Statistics

	Group 1		Group 2	
	Inflection	Acc. vs No-Acc.	Inflection	Acc. vs No-Acc.
Atoms	6	35	5	23
Behaviors	6741	972	1344	220

As you can see in table 1, there are some major differences in the number of atoms and behaviors identified between the *inflection* and *acceleration vs no-acceleration* segmentation methods. As can also be seen is the massive number of behaviors built.



(a) Course 1 Motions



(b) Course 2 Motions

Figure 23: Motion Paths taken in Experiment

The reason for this was discussed in chapter 5.2.2.2. So presented in table 2 are the reduced behavioral alphabets when only including those behaviors that were ever actually used by the final MDLe calls calculated in section 6.3.

Table 2: Alphabet Statistics with Used Behaviors

	Group 1		Group 2	
	Inflection	Acc. vs No-Acc.	Inflection	Acc. vs No-Acc.
Atoms	6	35	5	23
Behaviors	904	568	93	112

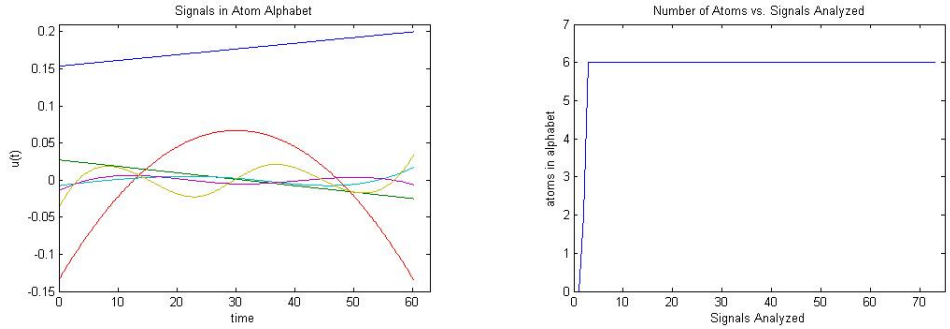
6.2.1 Atom Alphabet Construction Results

The construction of the atom alphabets was a very smooth process. The average processing time to build the atom alphabets was approximately 3 minutes per alphabet. We will now review the metrics presented in chapter 5.2.2.1.

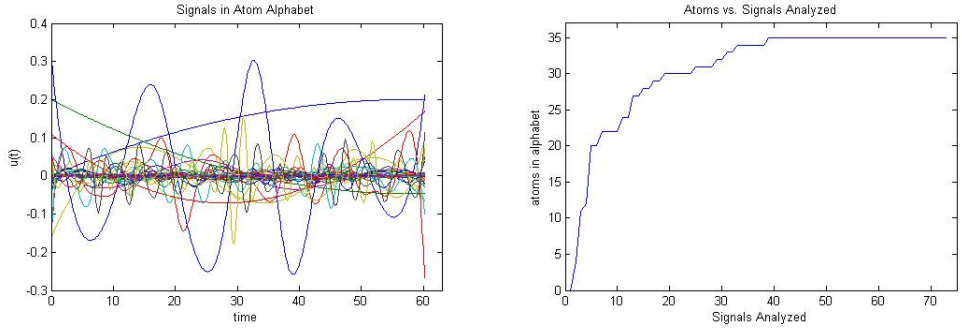
6.2.1.1 General Results

Some detailed graphs of the atom alphabets constructed from groups 1 and 2 are given in figures 24 and 25. In each set of figures for the groups, there is a set of graphs (24(a),25(a)) detailing the alphabets built using the *inflection point* method of temporal segmentation, as well as a set of graphs (24(b),25(b)) using the *acceleration vs no-acceleration* method of temporal segmentation.

For each method of temporal segmentation, there are two graphs given. The graph on the left shows all of the signals in the atom alphabet, while the graph on the right shows how the number of atoms in the alphabet grew as the algorithm analyzed the signals from each group. An interesting thing to note as that the number of atoms in the alphabets max out after analyzing a certain number of signals, which leads us to suspect that after a certain dimension, the alphabet contains enough bases to replicate all the signals. And if the alphabets are maxing out, then perhaps they

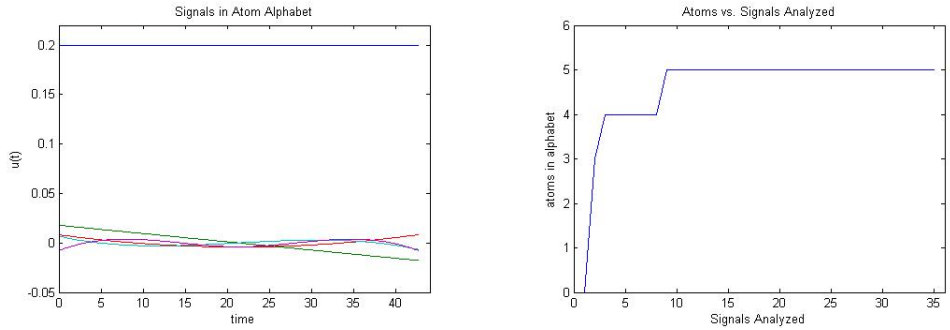


(a) Using Inflection Point Method of Temporal Segmentation

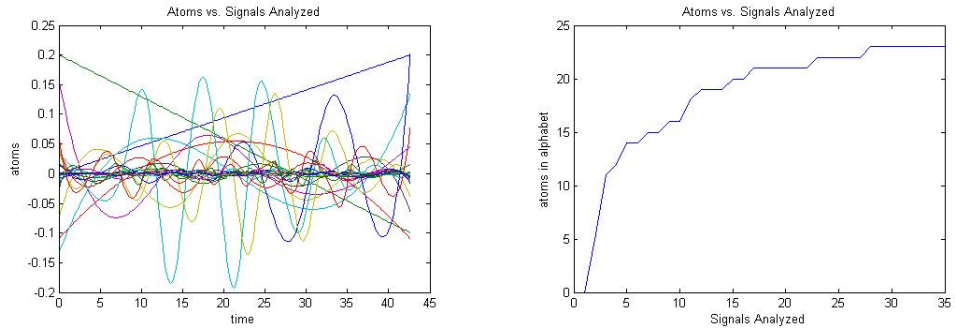


(b) Using Acceleration vs No-Acceleration Method of Temporal Segmentation

Figure 24: Atom Alphabet Stats for Group 1



(a) Using Inflection Point Method of Temporal Segmentation



(b) Using Acceleration vs No-Acceleration Method of Temporal Segmentation

Figure 25: Atom Alphabet Stats for Group 2

are all in fact the same subspaces. This is what we intend to check in the similarity section.

6.2.1.2 Atom Similarity Results

Shown in table 3 are the overlap and similarity ratios (%), as defined in chapter 5.2.2.1.

Table 3: Atom Overlap and Similarity Statistics

(a) Atom Overlap Stats

$X \subseteq Y$		Group 1		Group 2	
		Inflection (6 atoms)	Acc. (35 atoms)	Inflection (5 atoms)	Acc. (23 atoms)
Group 1	Inflection (6 atoms)	**100 %**	100 %	**83 %**	100 %
	Acc. (35 atoms)	14 %	**100 %**	9 %	**40 %**
Group 2	Inflection (5 atoms)	**100 %**	100 %	**100 %**	100 %
	Acc. (23 atoms)	17 %	**91 %**	17 %	**100 %**

(b) Atom Similarity Stats

Similarity		Group 1		Group 2	
		Inflection (6 atoms)	Acc. (35 atoms)	Inflection (5 atoms)	Acc. (23 atoms)
Group 1	Inflection (6 atoms)	**100 %**	17 %	**83 %**	26 %
	Acc. (35 atoms)	14 %	**100 %**	8 %	**32 %**
Group 2	Inflection (5 atoms)	**83 %**	14 %	**100 %**	22 %
	Acc. (23 atoms)	16 %	**57 %**	17 %	**100 %**

As is evident in table 3(a), the *inflection* alphabets from each group are almost exact copies of each other. Further, these inflection alphabets are complete subsets of each group's alphabet defined by the *acceleration vs no-acceleration* method.

On the other hand, the 2 alphabets defined by the *acceleration vs no-acceleration* method have somewhat disappointing results at first glance. But we can notice that they are of substantially different sizes, and so the similarity results are rightfully skewed as this ratio is based on the dimension of their intersection to the union of the two spaces. But if you look at the overlap of the smaller alphabet $G2$ with the larger alphabet $G1$, we see that 91% of $G2$ is contained within $G1$, and so it is almost

a subset of the larger alphabet $G2$.

A point of concern with these stats is evident in the similarity stats of table 3(b). We would expect this table to be symmetric, but unfortunately is not, especially with the *acceleration vs no-acceleration* alphabets. The reason for this is in the ϵ value chosen for the numerical projection test of equation (19). So in future versions of the algorithm, perhaps a method of normalizing the projecting vector and space such that the value of ϵ is not as important could be developed.

6.2.1.3 Behavioral Similarity Results

Shown in tables 4 and 5 are the number of behaviors defined in the full and reduced behavioral alphabets for each group, as well as the number of dimensions spanned by the reconstructed behavioral time signals as explained in chapter 5.2.2.2;

Table 4: Dimensions of $L_2[0,T]$ Space Spanned by Full Behavioral Alphabets

	Number of Behaviors		Dimensions of Space	
	Inflection	Acc. vs No-Acc.	Inflection	Acc. vs No-Acc.
Group 1	6741	972	84	55
Group 2	1344	220	52	38

Table 5: Dimensions of $L_2[0,T]$ Space Spanned by Reduced Behavioral Alphabets

	Number of Behaviors		Dimensions of Space	
	Inflection	Acc. vs No-Acc.	Inflection	Acc. vs No-Acc.
Group 1	904	568	61	48
Group 2	93	112	11	25

The first thing we notice is that the number of behaviors actually used by the MDLe$_b$ calls defined in the next section 6.3 is much less than those originally defined. The reasoning for this was discussed in section 5.2.2.2.

A second thing we see is that the $L_2[0,T]$ subspaces spanned by the behaviors is substantially smaller in dimension than the number of behaviors defining it. This is

an expected outcome though as the behaviors found by the algorithm were not meant to define a space like the atom alphabet. Rather, they just represent a segment from the group's original signals that occurred multiple times.

A final thing to note is the huge number of behaviors defined by the inflection alphabets, even though they had the smallest atom alphabets. This is especially evident in Group 1, which allowed the users to be as free form as possible.

With these behavioral spaces, we then performed the overlap and similarity analysis on them, as we did for the atom alphabets in 6.2.1.2. These results are shown in tables 6 and 7.

Table 6: Overlap and Similarity Statistics for $L_2[0,T]$ Space of Full Alphabets

(a) Full Behavior Overlap Stats

$\subseteq$		Group 1		Group 2	
		Inflection (84 atoms)	Acc. (55 atoms)	Inflection (52 atoms)	Acc. (38 atoms)
Group 1	Inflection (84 atoms)	**100**	40	**35**	11
	Acc. (55 atoms)	95	**100**	65	**31**
Group 2	Inflection (52 atoms)	**92**	44	**100**	27
	Acc. (38 atoms)	97	**68**	71	**100**

(b) Full Behavior Similarity Stats

$\subseteq$		Group 1		Group 2	
		Inflection (84 atoms)	Acc. (55 atoms)	Inflection (52 atoms)	Acc. (38 atoms)
Group 1	Inflection (84 atoms)	**100**	32	**27**	8
	Acc. (55 atoms)	60	**100**	51	**22**
Group 2	Inflection (52 atoms)	**55**	27	**100**	18
	Acc. (38 atoms)	44	**39**	43	**100**

What we can see is that the behavioral alphabets defined by the *inflection* method are much more all encompassing than those defined by the *acceleration vs no-acceleration* method, especially in terms of the original full behavior sets defined by the algorithm. We also see that the amount of overlap and similarity between the behavioral spaces reduces pretty drastically when we only look at those behaviors used by the MDLe_b

Table 7: Overlap and Similarity Statistics for $L_2[0,T]$ Space of Reduced Alphabets

(a) Reduced Behavior Overlap Stats

$\subseteq$		Group 1		Group 2	
		Inflection (61 atoms)	Acc. (48 atoms)	Inflection (11 atoms)	Acc. (25 atoms)
Group 1	Inflection (61 atoms)	**100**	34	**0**	0
	Acc. (48 atoms)	75	**100**	0	**6**
Group 2	Inflection (11 atoms)	**73**	73	**100**	27
	Acc. (25 atoms)	72	**64**	8	**100**

(b) Reduced Behavior Similarity Stats

$\subseteq$		Group 1		Group 2	
		Inflection (61 atoms)	Acc. (48 atoms)	Inflection (11 atoms)	Acc. (25 atoms)
Group 1	Inflection (61 atoms)	**100**	24	**0**	0
	Acc. (48 atoms)	49	**100**	0	**4**
Group 2	Inflection (11 atoms)	**13**	16	**100**	9
	Acc. (25 atoms)	26	**28**	6	**100**

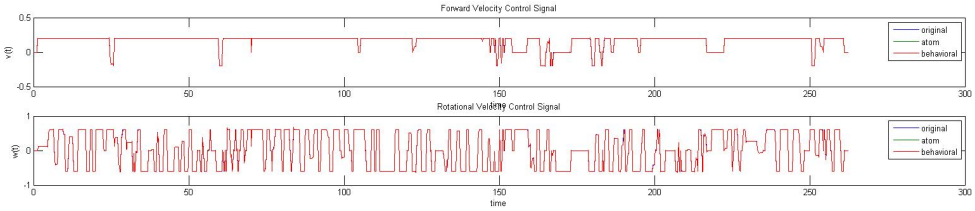
calls of section 6.3.

6.3 MDLe Strings

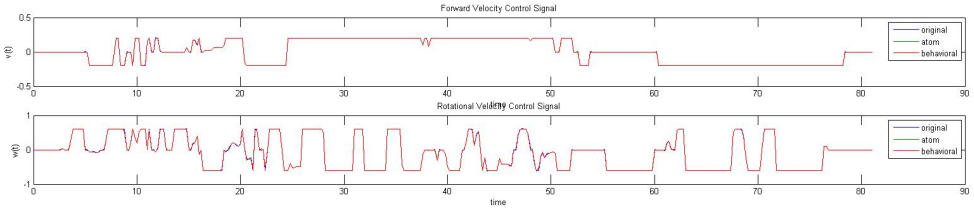
With the alphabets constructed, we then decomposed all of the signals from both groups into MDLe_a and MDLe_b calls based on that group's alphabets, as well as the alphabets of the other group. We begin by looking at the performance metrics defined in section 5.2.3. We then look at some detailed statistics on the resulting data transmission ratios of the various MDLe calls versus the original signal.

6.3.1 G1 Data Set

This section presents the results of the MDLe strings constructed for Group 1, which ran the course shown in figure 22(a). Two sample graphs of the forward and rotational velocity control signals are shown in figure 26. Each of these graphs is an overlain portrait of the user's original signals, and the reconstructed MDLe_a and MDLe_b signals.



(a) Sample Control 1



(b) Sample Control 2

Figure 26: Group 1 MDLe Sample Controls

6.3.1.1 Forward Velocity Control Signal

Given in table 8 are the performance statistics of how the MDLe calls constructed for the forward velocity control signals from Group 1 performed. The first sub-table 8(a) shows the performance of the MDLe_a and MDLe_b calls based on its own alphabets. The second sub-table 11(b) show the performance of the MDLe_a and MDLe_b calls based on the other group's alphabets.

With respect to the temporal segment periods, we see that they are on the scale of a few seconds, with a large standard deviation, which is good because it means that we are not switching between MDLe calls many times per second. This result is to be expected. When running the experiment, I noticed that many users would jam the forward velocity to max because the output was limited to a maximum of about 2-3 inches per second, a relatively slow pace. This behavior is exhibited in figure 26, in the form of very long constant signals at the maximum signal level.

Another observation on the temporal segments is that the average length always increased between the MDLe_a and MDLe_b calls, meaning that our behaviors are replacing sequences of atom segments with a single behavioral segment.

Table 8: G1 Forward Velocity MDLe Stats

(a) Based on G1 Alphabets

	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	1.97 s	4.61 s	3.18 s	4.50 s
Std	5.34 s	8.28 s	6.45 s	7.44 s
Error Ratio	0.016 %	0.016 %	0.050 %	0.050 %
Bytes Ratio	29.5 %	8.5 %	27.3 %	14.1 %

(b) Based on G2 Alphabets

	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	1.51 s	3.00 s	2.43 s	2.84 s
Std	4.09 s	7.33 s	4.93 s	5.23 s
Error Ratio	0.017 %	0.017 %	0.051 %	0.051 %
Bytes Ratio	19.1 %	12.8 %	23.7 %	16.2 %

Finally, we see that the MDLe$_a$ strings based on the *inflection* alphabets had much shorter segment periods than their *acceleration vs no-acceleration* strings. But when behaviors were used, the segment length became longer than the *acceleration vs no-acceleration* strings, which backs our earlier observation that the behavioral alphabets based on the *inflection* method were more all-encompassing than their *acceleration vs no-acceleration* counterparts.

With respect to the error ratio, we see that it is incredibly low, meaning that the MDLe calls are creating near perfect replications of the original signal, even when based on the alphabets from the other group. This can also be seen in figure 26, in that it is nearly impossible to visually discern the differences between the original signals and MDLe calls. Further, we see that there is no difference in error ratios between the MDLe$_a$ and MDLe$_b$ calls, which makes sense because the behavior calls are based wholly on the atom calls they are replacing.

And in terms of the *inflection* versus *acceleration vs no-acceleration* methods, we see that the *inflection* method is slightly more accurate.

Finally, with respect to the bytes ratio, we see that there is a massive drop in the number of bytes required to transmit the MDLe calls versus the original signal. More on this will be discussed in section 6.3.3.

6.3.1.2 Rotational Velocity Control Signal

Given in table 9 are the performance statistics of how the MDLe calls constructed for the rotational velocity control signals from Group 1 performed.

Table 9: G1 Rotational Velocity MDLe Stats

(a) Based on G1 Alphabets

	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	0.38 s	0.97 s	0.90 s	1.27 s
Std	0.95 s	1.69 s	1.26 s	1.50 s
Error Ratio	0.082 %	0.082 %	0.173 %	0.173 %
Bytes Ratio	72.6 %	25.3 %	64.7 %	43.6 %

(b) Based on G2 Alphabets

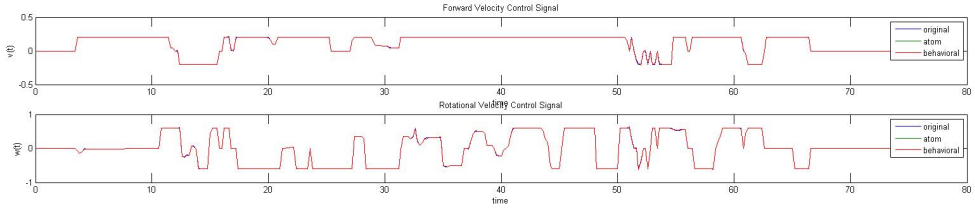
	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	0.29 s	0.57 s	0.68 s	0.82 s
Std	0.73 s	1.17 s	0.96 s	1.02 s
Error Ratio	0.074 %	0.074 %	0.222 %	0.222 %
Bytes Ratio	49.3 %	35.6 %	60.3 %	46.6 %

As opposed to the forward velocity signals, the segment periods for the rotational velocity signals are much shorter. This is to be expected because the users were forced to adjust the orientation of the Khepera much more often than the forward velocity. This can be seen visually in figure 26. But like the forward velocity signals, we can observe an increase in segment period of the MDLe_b strings over the MDLe_a calls, as well as longer segments for the *acceleration vs no-acceleration* strings over the *inflection* strings.

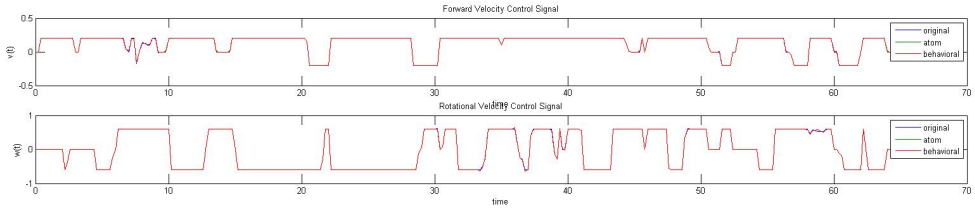
We also see large decreases in the error ratio and bytes ratio, regardless of which group's alphabets are being used to construct the strings, which mirrors the results

we saw for the forward velocity controls. Further, we see that the *inflection* based strings are more accurate than the *acceleration vs no-acceleration* strings.

6.3.2 G2 Data Set



(a) Sample Control 1



(b) Sample Control 2

Figure 27: Group 2 MDLe Sample Controls

This section presents the results of the MDLe strings constructed for Group 2, which ran the course shown in figure 22(b). Two sample graphs of the forward and rotational velocity control signals are shown in figure 27. Each of these graphs is an overlain portrait of the user's original signals, and the reconstructed MDLe_a and MDLe_b signals.

The results for Group 2 are almost indiscernible from the results of Group 1. All of the same conclusions relating the performance of strings based on its own alphabet versus the others alphabet, the comparison between MDLe_a and MDLe_b calls, and the differences between the *inflection* and *acceleration vs no-acceleration* strings can be made.

The only noticeable differences can be attributed to the difference in the tasks performed by Group 1 and Group 2. Group 2 had a much more structured task, and so we would expect the signals generated to be much less chaotic than those from

Group 1, which might generate longer temporal segments, leading to a better bytes ratio.

6.3.2.1 Forward Velocity Control Signal

Table 10: G2 Forward Velocity MDLe Stats

(a) Based on G1 Alphabets

	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	2.10 s	5.16 s	3.38 s	4.51 s
Std	5.39 s	7.78 s	6.47 s	7.33 s
Error Ratio	0.010 %	0.010 %	0.051 %	0.051 %
Bytes Ratio	26.5 %	9.7 %	24.8 %	13.3 %

(b) Based on G2 Alphabets

	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	2.10 s	3.58 s	3.38 s	4.42 s
Std	5.39 s	8.80 s	6.47 s	7.23 s
Error Ratio	0.020 %	0.020 %	0.051 %	0.051 %
Bytes Ratio	17.1 %	11.6 %	20.8 %	13.3 %

Given in table 10 are the performance statistics of how the MDLe calls constructed for the forward velocity control signals from Group 2 performed. The first sub-table 10(a) shows the performance of the MDLe_a and MDLe_b calls based on the other group's alphabets. The second sub-table 10(b) shows the performance of the MDLe_a and MDLe_b calls based on its own alphabets.

As mentioned before, we see all of the same general characteristics found in Group 1's data set. The only difference is that on average, the temporal segments are longer than those found in the forward velocity signals of Group 1.

Table 11: G2 Rotational Velocity MDLe Stats

(a) Based on G1 Alphabets

	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	0.50 s	1.14 s	1.02 s	1.35 s
Std	1.20 s	1.94 s	1.47 s	1.61 s
Error Ratio	0.042 %	0.042 %	0.154 %	0.154 %
Bytes Ratio	60.5 %	22.1 %	54.1 %	37.9 %

(b) Based on G2 Alphabets

	Inflection		Acc	
	$MDLe_a$	$MDLe_b$	$MDLe_a$	$MDLe_b$
Segment Period	0.50 s	1.00 s	1.02 s	1.26 s
Std	1.20 s	2.03 s	1.47 s	1.58 s
Error Ratio	0.065 %	0.065 %	0.144 %	0.144 %
Bytes Ratio	39.9 %	27.1 %	50.0 %	36.6 %

6.3.2.2 Rotational Velocity Control Signal

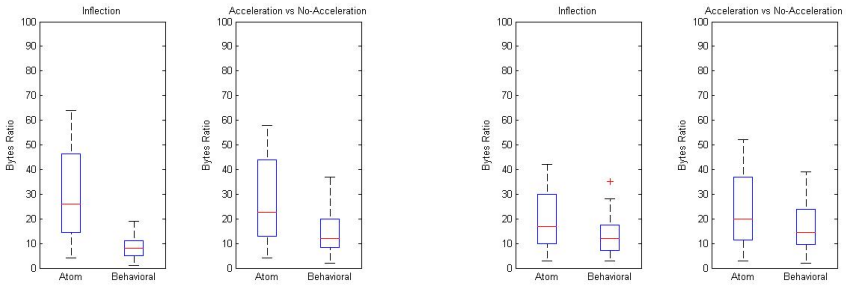
Given in table 11 are the performance statistics of how the MDLe calls constructed for the rotational velocity control signals from Group 2 performed. The first sub-table 11(a) shows the performance of the MDLe$_a$ and MDLe$_b$ calls based on the other group's alphabets. The second sub-table 11(b) shows the performance of the MDLe$_a$ and MDLe$_b$ calls based on its own alphabets.

6.3.3 Data Transmission Statistics

In this section I present the data transmission statistics between the MDLe calls and the original control signals. We have already seen the average ratios in the Group 1 and Group 2 performance sections. But we will now present some detailed box and whisker plots containing the ratios of every single MDLe string constructed.

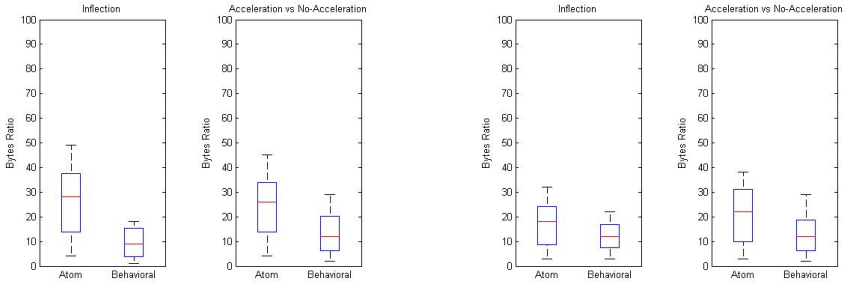
6.3.3.1 Forward Velocity Controls

Given in figures 28 and 29 are the box and whisker plots of the bytes ratio for the forward velocity control signals from Groups 1 and 2.



(a) Against G1 Alphabet (b) Against G2 Alphabet

Figure 28: Bytes Transmitted Ratio for Group 1, forward velocity control



(a) Against G1 Alphabet (b) Against G2 Alphabet

Figure 29: Bytes Transmitted Ratio for Group 2, forward velocity control

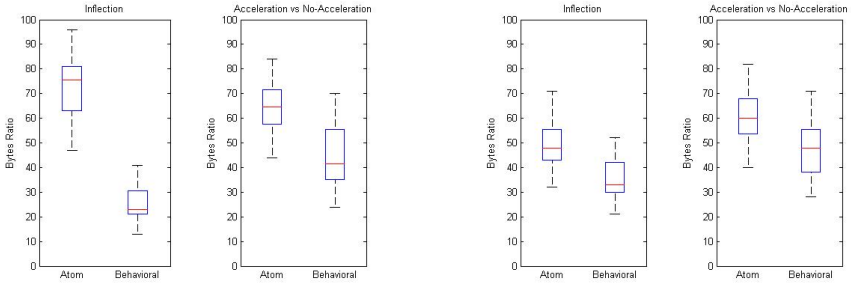
We note that the ranges of all the MDLe string types are in the $[1\%, 50\%]$, and are generally comparable. But one thing to notice is that the strings based on Group 2's alphabets have MDLe_a and MDLe_b transmission ratios that are generally lower than those same calls made against Group 1's alphabet. An exception to this is the huge decrease when making MDLe_b calls based on Group 1's inflection alphabet, which has an extremely low ratio of about $10\% \pm 5\%$.

6.3.3.2 Rotational Velocity Controls

Given in figures 30 and 31 are the box and whisker plots of the bytes ratio for the rotational velocity control signals from Groups 1 and 2.

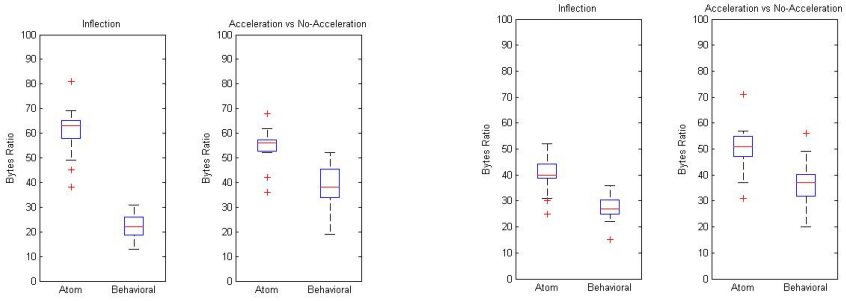
What we see is that the range of decrease in data transmission is on average much higher than the forward velocity controls. This can be attributed to the increased complexity of the rotational velocity signals.

A notable performance is again shown by the MDLe_b strings based on Group 1's inflection alphabet, which produces a massive decrease over the MDLe_a strings, and secures the lowest data transmission rates of all the alphabets.



(a) Against G1 Alphabet (b) Against G2 Alphabet

Figure 30: Bytes Transmitted Ratio for Group 1, rotational velocity control



(a) Against G1 Alphabet (b) Against G2 Alphabet

Figure 31: Bytes Transmitted Ratio for Group 2, rotational velocity control

CHAPTER VII

CONCLUSION

In examining the experimental results of chapter 6, we have an answer to our question of whether bottom-up mimicry can be used to generate all-encompassing control alphabets. This answer is a resounding yes. All of the alphabets we generated proved capable of reproducing the controls of the other group with essentially no error. Thus, any one of these alphabets could be stored on a robotic apparatus, and be capable of enacting the high-level tasks a user desires.

But our results also present a bit of a conundrum. Here we have four distinct alphabets constructed in different manners that were all capable of recreating the controls with almost no error. So which method of alphabet generation is the best?

The answer looks like the inflection method generated by a group of freewheeling users, like those in Group 1. This alphabet had the lowest error ratios of all the alphabets. Further, it generated the largest number of behaviors, allowing it to generate some of the longest temporal segments, leading to dramatic reductions in required bandwidth.

In fact, we saw in the overlap statistics that it was completely contained within the acceleration vs no-acceleration alphabets, and was a super-set to Group 2's inflection alphabet. So perhaps, this little 6-dimensional space was the powerhouse behind all of the other alphabets. But who knows. We would need to run a more targeted experiment to claim this with any certainty.

And so we conclude this thesis with a quick discussion on future steps that can build off of this work.

7.1 Future Work

The next major step to take with this work is to bring these MDLe alphabets back into the mind set of MDLe. What we've done in this thesis is developed a method of creating a set of all-encompassing signals that can be stored on a robotic apparatus, and called using the language of MDLe. But we did not develop a concrete mapping between the high-level tasks that generated the alphabets, and the MDLe calls that represent them. So unless this is done, all we have is a powerful set of control signals, with no system to use them.

One method of creating these maps would be to generate a program in which the high-level user defines and performs a task, while the system records which combination of signals create the defined task. This is simple, and would allow the construction of these maps.

But a more interesting direction to take is to stay within this style of bottom-up construction, and refrain from forcing the user to define their own tasks, because they may not have the insight to define the fundamental tasks they are actually performing. Instead, what if we created a learning algorithm that could watch users performing their tasks, and learn the true tasks they were performing. We could then map these tasks to the exact MDLe strings that produce the controls.

In relation to our experiment, such a system might make a guess as to the current task being performed based on the current and recent actions of the joystick, and generate a set of spatial and temporal scalars for the MDLe calls that are mapped to the task being performed.

Another thought is to insert a neural network between the high-level user and these MDLe alphabets. This neural network could work in real-time to translate the desired user actions into the MDLe calls that create them.

There are many routes we could take, but the next step needs to be the development of a mapping method to use these all-encompassing controls.

REFERENCES

[1] Andersson, S. and Hristu, D., "Symbolic feedback control for navigation," *Automatic Control, IEEE Transactions on*, vol. 51, no. 6, pp. 926–937, 2006.

[2] Bekey, G., *Autonomous robots: from biological inspiration to implementation and control*. The MIT Press, 2005.

[3] Brockett, R., "On the computer control of movement," in *Robotics and Automation, 1988. Proceedings., 1988 IEEE International Conference on*, pp. 534 –540 vol.1, apr 1988.

[4] Brockett, R., "Formal languages for motion description and map making," *Robotics*, vol. 41, pp. 181–193, 1990.

[5] Brockett, R., "Hybrid models for motion control systems," *PROGRESS IN SYSTEMS AND CONTROL THEORY*, vol. 14, pp. 29–29, 1993.

[6] Caines, P. and Wei, Y., "Hierarchical hybrid control systems: A lattice theoretic formulation," *Automatic Control, IEEE Transactions on*, vol. 43, no. 4, pp. 501–508, 1998.

[7] Hristu-Varsakelis, D., Egerstedt, M., and Krishnaprasad, P., "On the structural complexity of the motion description language mdle," in *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, vol. 4, pp. 3360–3365, IEEE, 2003.

[8] Li, H. and Brockett, R., "Dmdl for choreography," in *Decision and Control, 2005 and 2005 European Control Conference. CDC-ECC'05. 44th IEEE Conference on*, pp. 4016–4021, IEEE, 2005.

[9] Luenberger, D., *Optimization by vector space methods*, p. 56. Wiley-Interscience, 1997.

[10] Manikonda, V., Hendler, J., and Krishnaprasad, P., "Formalizing behavior-based planning for nonholonomic robots," in *International Joint Conference on Artificial Intelligence*, vol. 14, pp. 142–149, LAWRENCE ERLBAUM ASSOCIATES LTD, 1995.

[11] Manikonda, V., Krishnaprasad, P., and Hendler, J., "A motion description language and a hybrid architecture for motion planning with nonholonomic robots," in *Robotics and Automation, 1995. Proceedings., 1995 IEEE International Conference on*, vol. 2, pp. 2021 –2028 vol.2, may 1995.

[12] Manikonda, V., Krishnaprasad, P., and Hendler, J., *Mathematical control theory*, ch. Languages, behaviors, hybrid architectures and motion control, pp. 199–226. Springer Verlag, 1999.

[13] Martin, P. and Egerstedt, M., "Optimal timing control of interconnected, switched systems with applications to robotic marionettes," in *Discrete Event Systems, 2008. WODES 2008. 9th International Workshop on*, pp. 156–161, IEEE, 2008.

[14] Martin, P. and Egerstedt, M., "Optimization of multi-agent motion programs with applications to robotic marionettes," *Hybrid Systems: Computation and Control*, pp. 262–275, 2009.

[15] Martin, P. and Egerstedt, M., "Expanding motion programs under input constraints," in *American Control Conference (ACC), 2010*, pp. 2416–2421, IEEE, 2010.

[16] Martin, P. and Egerstedt, M., "Timing control of switched systems with applications to robotic marionettes," *Discrete Event Dynamic Systems*, vol. 20, no. 2, pp. 233–248, 2010.

[17] Martin, P., Johnson, E., Murphy, T., and Egerstedt, M., "Constructing and implementing motion programs for robotic marionettes," *Automatic Control, IEEE Transactions on*, no. 99, pp. 1–1, 2011.

[18] Mehta, T. and Egerstedt, M., "An optimal control approach to mode generation in hybrid systems," *Nonlinear analysis*, vol. 65, no. 5, pp. 963–983, 2006.

[19] Mehta, T. and Egerstedt, M., "Optimal membership functions for multimodal control," in *American Control Conference, 2006*, pp. 6–pp, IEEE, 2006.

[20] Zhang, W. and Tanner, H., "Composition of motion description languages," *Hybrid Systems: Computation and Control*, pp. 570–583, 2008.